STATIC ANALYSIS FOR ARCHITECTURE-IMPLEMENTATION CONFORMANCE IN ROBUST EMBEDDED SYSTEMS

BY

SUMANT JAGADISH KOWSHIK

B.Tech., Indian Institute of Technology, Madras, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Embedded systems have proliferated into diverse and complex critical applications with stringent reliability and timeliness requirements. Guaranteeing reliability in the presence of increasing complexity of embedded systems have necessitated a multitude of architectural designs including integrated modular architectures and architectural designs for robustness by minimizing inter-component failure dependencies.

In the software development cycle, the system integration architect occupies a key position between the domain-specialist, designing the algorithms and the high-level logical design, and the individual software component developers. In essence, the system architect refines the logical design into concrete software components, while facilitating high-level properties such as timing, dependency management, and fault-tolerance. Existing tools for the systems architect include architecture description languages and model-checking tools, which specify and verify the architectural designs.

However, there is a gap between the architectural principles and the actual implementations developed by individual software developers. Low-level software errors, particularly in languages like C and C++, such as dangling pointer dereferences and array bounds errors, violate architectural properties. Recent research on debugging tools focus on best-effort approaches to detecting low-level programming errors. However, there is a need for tools that guarantee that high-level architectural properties are enforced in the component implementation. Failure to verify these properties in the actual code have caused two critical disasters in recent years, the satellite Phobos I and the Ariane V rocket.

The primary contribution of this dissertation is to design a system to analyze individual components and guarantee high-level architectural properties in the system using static analysis. In particular, we verify two key properties: (a) memory isolation and (b) safe value propagation paths from non-core to core components communicating using shared memory. Our solution combines language and library usage restrictions on the C language with a suite of compiler analyses to statically guarantee these properties. In doing so, we incur minimal (often zero) run-time overhead and do not re-

quire garbage collection, making our approach very attractive for embedded systems. We have examined different critical systems and embedded benchmarks and shown that our language restrictions are expressive enough for embedded systems while enabling statically guaranteeing high-level architectural properties. Finally, we show that we can verify other related architectural properties by extending our static analysis techniques.

To Sri Sathya Sai Baba
To Amma, Anna, and Shobha

Acknowledgements

I owe this dissertation and all the learning that accompanied it to “a few very good men”. My advisor Prof. Sha provided the motivation for the research content in this work. More importantly, his emphasis on research methodology and true learning, along with the incredible flexibility he provided me in my research has given me confidence and perspective for anything I undertake hereon. I have learned more about the virtues of being a doctoral student from Dr. Vikram Adve than anybody else. He taught me how to think, speak, and write. His passion for compilers in particular and computer science in general is undoubtedly contagious. I would also like to thank Dr. Grigore Rosu and Prof. P. R. Kumar, both of whom have been generous with their time providing excellent discussions and interesting ideas, a fraction of which, I hope, I have expanded upon in this dissertation.

I was very fortunate to draw inspiration and encouragement when the chips were down from my parents and Shobha. My parents have led by example and made great sacrifices to enable this milestone. Only their (and my grandparents') patience, spirit, and affection have made this moment possible. Over the last two years, while I negotiated the Ph.D alleyways, Shobha has served the very contrasting roles of being both a fulcrum and a source of unending fun. The entertainment, analysis, love, and wisdom led to priceless moments, speedy recoveries from troughs, self-discovery (perhaps!), and happiness in its purest forms. But what lingers in my mind most prominently is the inspiration of a person who recognizes ideals and, then, embodies them. Not least, in the realm of the mind.

I would be remiss if I did not acknowledge the contribution of a few great friends: Dinakar and Chris, for much assistance with the research, my brothers Prashant and Hemant for keeping me sane; Pappu (particularly!), Sathish (especially!), Sam, Jey, Sandy, and Ravikant for enlivening each day and constantly broadening the expanses of my mind; the RTSL crew with Ajay, Kihwal (the “systems” guy), Liu, Hui, Tanya, and Qixin for being great compatriots. Finally, I will be forever indebted to UIUC and the CS department, for it is here that I grew up and learned to think, work, play, and serve society. I shall cherish the generosity of spirit that invites people from faraway lands to enjoy these opportunities in the USA.

Om Sri Sai Ram.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The proliferation of embedded systems and control systems constitute the next step in the information technology revolution [46], where a network of computers interact with the physical world. We are witnessing widespread deployment of devices at homes, offices and critical infrastructure systems such as power plants and aircrafts. Evolving from being software running on small hardware devices, embedded systems are being networked into hierarchical distributed systems co-ordinating complex and critical applications. Unlike desktop or server software, the interaction with the physical world entails that embedded systems guarantee a core subset of critical requirements, termed *safety requirements*.

Traditionally, simple embedded systems were designed by a domain expert, who started with a simple design and computed the set of equations that model the physical system. For instance, a control system would be designed using three components, sensor, controller, and actuator, to receive inputs from the physical system, process control outputs, and send the control outputs to the physical system respectively. Most of the effort at the design stage is focused on developing controllers that accurately model the physical system. Upon finalization, this components in the design were directly translated to individual software components developed by software developers. In this design process, the system was only as reliable as its most complex component, the controller.

Achieving reliability in modern embedded systems is, however, an increasingly challenging goal due to growing complexity. The demand for functionality of embedded systems has led to a rapid increase in the complexity of embedded software. Increasing complexity leads to a higher probability of programmer errors, errors due to integration such as timing errors, etc. Traditional tools for enforcing reliability of components such as rigorous testing and manual inspection are intractable. Moreover, a single fault in a complex system tends to bring down the entire system functionality including the critical functionality of the system. As a result, complexity due desirable features such as usability, performance and other secondary concerns can compromise the primary, critical functionality i.e system safety. This concept is captured by the upgrade paradox [83] in re-

dundant systems where an untested software upgrade, when installed, can potentially bring down the entire system, and yet cannot be tested without installation. Growing complexity necessitates software architectural designs that partition features among different components, track dependencies between components, and provide security, fault containment and fault-tolerance.

1.1 System Integration Architecture

The need for architectural design solutions for robustness has led to the emergence of a system integration architect in the embedded system development timeline. The role of the integration architect is to design the non-functional requirements of a system such as robustness and security. As noted above, these architectural requirements are increasingly difficult, if not impossible, to provide in simple and monolithic system designs. In this dissertation, we focus only on architectural solutions for robustness. Other non-functional properties such as security are extensions to this work and are discussed in chapter 4.

Being unable to avoid faults completely, recent work on building reliable systems has focused on *fault-tolerance* and *fault containment* [82, 81, 79, 16, 41]. Internally partitioning functionality into levels depending on their criticality protects the components providing critical functionality from relatively untested and unreliable software providing non-critical (though desirable) functionality. In laboratory-scale control systems, we observe that there are typically three levels of functionality: (1) safety features, which guarantee a core subset of features, (2) performance, and (3) user-interfaces and bookkeeping. We illustrate the system design to incorporate this feature separation in section 1.7.

Simplex [82, 81] is an architectural framework that enables safe online upgrade of controller software. The architecture describes a stability envelope based on the Lyapunov stability envelope within the operationally admissible states of a controlled plant (e.g., an airplane or an inverted pendulum in the lab), which protects the system against algorithmic or value errors in the upgraded, untested controller, immediately switching back to a safe well-tested and reliable (though low-performance) controller when the plant is detected to be outside the stability envelope. In addition to value errors, Simplex also protects against timing errors and errors detectable at run time such as crashes, stack or heap overflow or divide by zero. In practice, simplex assumes that the components have memory and logical system resource sandboxes thus preventing a memory or a logical system resource corruption of a peer component.

Examining the architectural designs for robustness, a key principle that

emerges is the following:
Separation Principle: Logical and/or physical separation of core components (which implement critical functionality) and non-core components such that a fault in the non-core subsystem *cannot* compromise the core subsystem functionality.

Informally, the separation principle adumbrates that the *core subsystem does not depend on the non-core subsystem, though it can safely use its functionality*. The formal definition of the relationships *use* and *depend* between components are described in [34], based on the effect of faults in non-core components on the core subsystem. Essentially, if any fault in a non-core component propagates to a core functionality failure, the core component is said to *depend* on the non-core component, which violates the separation principle. The definition has three main implications, the first two of which are adapted from Rushby [79]:

1. [SP1] **Spatial Separation** : enforces that a non-core component cannot alter the private data or code of a core component, nor command the private devices or actuators of the other components.

2. [SP2] **Temporal Separation** : ensures that the resources received by a core component from a shared platform cannot be affected by a non-core component's actions. This includes processing, memory, and other logical system resources such as file descriptors and communication bandwidth.

3. [SP3] **Safe Inter-component Interaction**: enforces that any data received by the core component from a non-core component is (a) received along known channels and (b) monitored for safety before being used by the core component.

Many sub-properties of the separation principle have been previously specified as an architectural design requirement, particularly in the context of avionics architectures by Rushby [79], who has described a detailed list of properties that integrated modular architectures (IMAs) need to satisfy for high-assurance applications. However, the definition above describes component isolation from a robustness perspective, which is a contribution of this dissertation and other related work [34].

In existing state of the art designs, the properties SP1, SP2, and SP3 are supported by architectural design tools and system utilities. Sophisticated architectural design tools such as architecture description languages (ADL's) are formal tools which are aided by model-checking and design verification technology. Significantly, the SAE-AADL (Architecture Analysis and Design Language) [77] has emerged as a standard for avionics and embedded systems development. Steve Vastal has proposed an error model annex [77] to AADL, which is a language that specifies and analyzes faults

and their propagation at the design level. Other tools that aid design and verification of architectural designs are described in section 1.6. Each of these tools operate on designs, which are modular abstractions of the software. These designs are used as documentation or as models from which code stubs are generated. The component developers code the actual implementation of the individual components.

System utilities can also be exploited by the system integration architect in order to guarantee the three properties above. For instance, SP1 can be guaranteed through address space protection, if present on a certain platform, by implementing a core component and a non-core component as separate processes (tasks). This causes direct memory violations by a non-core component to result in a segmentation fault-like run-time error, thus protecting the core component. More recently, powerful paradigms such as light-weight real-time virtual machines [66, 67] are used to sandbox each component in a system to a fixed number of system resources.

1.2 Architecture-Implementation Gap

While, architectural designs and the accompanying verification tools have addressed design verification, these tools do not verify that the actual implementation conforms to the architectural specifications. Implementation of components are prone to diverse programmer errors, which can violate the assumptions or the specifications of the architectural design. Broadly, the implementation errors violate architectural properties in two ways: (a) By using erroneous programming constructs leading to arbitrary errors and (b) By violating the architectural specification along some path/s in the program. Until recently, there had been little effort in either exhaustive checking the implementation for common errors or to embed semantic information that is updated with the evolution of the program, which can be used to verify the conformance of the implementation to the architectural design specifications. In other words, there is a need for tools which verify that there is no divergence between the architectural design and the real system implementation.

The three properties, SP1, SP2, and SP3, comprising the separation principle described in the previous section, can be violated due to various implementation-level issues. For instance, SP1 and SP2 assume that core-components are protected against memory errors in non-core components as well as system resource errors. In other words, a memory error such as a C array overflow in a non-core component such as an unreliable controller may corrupt the memory of a core component such as safe controller, when there is no address protection between the two components. Even in the presence of address protection, buffer overflow attacks [3] can be employed

by the core component to execute illegal system calls from its data area, which hog system resources or even execute "lethal" system calls such as `kill -9`. These violations of the separation principle are basically at the implementation level and are not captured by architecture design verification tools or, in many cases, even system-level utilities.

Architectural design and assumption violation due to implementation errors have had catastrophic consequences in the past. The Ariane V explosion [36] was a result of an overflow in the velocity variable when it was read by a software component that only allocated two bytes to encode the velocity. Significantly, the component that had this bug provided a non-critical function, which propagated causing a failure in the core subsystem. This is a violation of the separation principle, which is a vital architectural requirement in such expensive and critical software programs. Similarly, in the Phobos I spacecraft, a keyboard buffer overflowed into the memory of a critical aircraft control function [19] corrupting a critical controller module.

1.2.1 Software Implementation Errors

Buggy software have been reported by multiple studies [42] to be the predominant source of unplanned downtime in large scale computing infrastructures. Fault-avoidance techniques such as system validation and analytical models [85] have resulted in valuable techniques for building fault-free software, but have also shown the impracticality of applying these techniques in large software systems. Candea *et. al* [42] postulate that while other branches of engineering build and maintain systems subject to laws of physics, software is only guided by rules laid down by the programmer, thus resulting in potentially unbounded failures.

Below, we list the means by which implementation errors can compromise the separation principle, hitherto established to be the most important architectural design for robustness. While this is by no means a comprehensive list, it is derived from a survey of errors in three different prototype systems (described in section 1.7), the simplex architecture [82], a car control testbed [46], and TinyOS [52]. Implementation errors in unreliable software components can compromise critical software functions through the following means:

1. Memory errors: Most embedded platforms and programming languages (C,C++) do not offer address or memory protection (through the MMU, for instance) thus making it possible for an unreliable component to corrupt the code or data of core components (violating SP1). On platforms that have address protection, memory errors can be used to execute arbitrary system calls from the data area, which can potentially corrupt the core components or starve them of resources (violating SP1 and SP2).

2. Logical system resource errors: An unreliable component can simply hog system resources such as memory and file descriptors (violating SP2)

3. The safe propagation of non-core components to the core components (SP3) could be violated in either of the following two ways:

   (a) Missing run-time monitors (e.g. system recoverability checks) for values propagated from non-core components to core components along some path in the core component.

   (b) Hidden dependencies due to inadvertent shared variables between non-core and core components, leading to unexpected propagation of unmonitored non-core values to the core component.

In addition, implementation errors can violate many other architectural requirements. For instance, controllers which can receive unreliable or potentially delayed feedback from sensors require that all incoming feedback values are monitored using a state estimator like a Kalman filter [21] (see chapter B for details) that quantifies the quality of the feedback. Absence of these filters for some feedback variable in the program can result in the core component using out-of-date information along some path in the implementation. Similar violations of security properties such as integrity and confidentiality [29] due to implementation errors can be constructed.

1.3 Architecture-Implementation Conformance

The broad goal of this work is to equip the system integration architect with tools that enable her to verify that the implementation of the system satisfies the architectural design principles. Thus, we attempt to bridge the gap between the architecture and the implementation. As a specific instance of architectural design specifications, we aim to verify that the components in the system satisfy the separation principle i.e., that no fault in a non-core component cause a core subsystem failure.

In general, properties SP1, SP2, and SP3 can be verified in the implementation by using a combination of static and dynamic analyses, system run-time utilities, and formal methods. In order to determine our approach, we examine some key properties of embedded programs/systems:

- First, most embedded programmers use weakly typed languages such as C or C++. This is a result of the performance-sensitivity of these applications, which makes the fine-grained control offered by C (and C++) very attractive.

- Embedded platforms are small and do not have the processing power or the memory of regular desktop or server platforms. Thus, these programs have stringent power, memory, and performance requirements. Recent work has shown that garbage collection (and by extension extensive run-time checking) increases power consumption by up to % [4].

- Embedded platforms often do not offer address space protection and is unable to prevent tasks from corrupting each other's data and code area. This has been coupled with the trend in embedded systems from a federated architecture to an integrated modular architecture [79], where a single computer system provides computing resources to several functions. The primary motivation for this architectural migration is the increasing hardware costs of providing individual computing systems for components in complex control systems such as in avionics. Traditional designs where software components run on dedicated computer systems led to diverse hardware making the certification of software components on unique platforms expensive.

- Detecting the existence of erroneous dependencies at run-time is particularly unattractive for critical embedded systems, for two reasons. First and foremost, detecting such architectural violations in critical systems may be "too late". The system may already be in an unrecoverable state potentially leading to catastrophic effects. Secondly, run-time exception handling is generally challenging to code, particularly in languages like C and C++. Weimer *et al* [94] found a large number of errors even in real-world Java [45] programs, where exception handling is arguably better supported by the language than C and C++. Moreover, run-time exception handling has a high performance penalty.

In addition to the above properties, it is desirable that our techniques for verifying the implementation are legacy code compatible. Also, any technique, which requires programmer assistance via annotations or code rewriting should incur minimal overhead on the programmer. With respect to annotations, Leveson [68] has demonstrated that programmers have varying ability in writing functional invariants and often wrongly place self-checks in their code. Hence, any annotations required of the programmer should be very simple, local, and easy-to-use.

1.4 Static Analysis for Architecture-Implementation Conformance

In this work, we demonstrate the utility and the advantages of static analysis in validating that the component implementation satisfy architectural design properties. In particular, we demonstrate the contribution of static analysis in validating the properties of the separation principle between the core and the non-core subsystem. We have developed two analyses:

1. Static analysis for memory safety: Statically guaranteeing that a component cannot overwrite any memory that is not allocated for or by it and that it cannot execute any code from its data area. In addition, it is enforced statically that each component only invokes trusted system call wrappers provided, preventing any system resource errors (the run-time checks themselves are outside the scope of this work).

2. Static analysis to check value propagation path errors: Statically guarantee that any value communicated through shared memory by a non-core component is monitored to satisfy system reliability (or recoverability) conditions before being used by a core component.

Both analyses rely on some support from system utilities, early run-time checks, formal methods or manual inspection. Specifically, our analysis to guarantee memory safety guarantees SP1 (spatial separation) with the aid of a run-time system that detects if the heap or stack space grows beyond a certain bound. The second analysis above verifies that all non-core values are monitored by the core component before being used (safe inter-component interaction, SP3). The correctness of the monitor itself is not guaranteed by the analysis and can be assured only by formal reasoning or manual inspection. In this sense, static analysis alone does not validate the separation principle, but is a significant component of a larger system that guarantees the properties. Moreover, static analysis minimizes the dependence on the run-time system, manual inspection or difficult-to-scale formal methods. Towards guaranteeing the separation principle, we rely on a run-time system to validate SP2 (temporal separation), though our analysis for memory safety contains a coarse-grained check that restricts the system calls invoked by a component to a safe subset.

Our basic approach involves starting with the C language and imposing a few semantic restrictions in order to facilitate static checking of the desired properties. The language restrictions reduce and in most cases eliminate the false positives generated due to the imprecision of static analysis. In our experiments, we verify that these semantic language rules are reasonable to implement a broad class of embedded and control systems. The

second analysis above also requires the programmer to insert a few annotations regarding semantic information such as the properties of the monitor in the core component.

Due to the aforementioned properties of embedded systems, compiler-based static analysis provides four distinct advantages over techniques such as testing, formal verification and dynamic checks. First, it is done at compile time, thus avoiding any run-time overhead, making it very attractive for embedded system. Secondly, it can provide guarantees by conservatively analyzing all the paths in the programs for a certain property. This distinguishes it from testing which is limited by its input set and the coverage achieved. Thirdly, it provides early error detection, enabling detecting many programming errors at compile time rather than during the execution of a critical system when failures can have a high performance penalty. Finally, compiler analyses examine the implementation instead of the architectural model, thus detecting any hidden dependencies on unreliable components. This provides guarantees on the implementation rather than the system design.

Static analysis has its limitations. Being a conservative analysis, false positives can result in reporting errors where none exist. In these cases, manual examination is required to actually check if the reported error exists. In addition complex language constructs such as complex array indices and arbitrary casting in C reduces the precision of the analysis, necessitating run-time checking of the properties in these cases. Static analysis is also ineffective in checking semantic properties that are not expressible by variables in the code or cannot be annotated on variables in the code. For dynamically changing properties, static analysis can only be used to verify that the required run-time checks are appropriately invoked.

1.5 Summary of Contributions and Limitations

The overall contribution of our work is to demonstrate that static analysis is an important tool for the system integration architect to verify high-level architectural properties in the implementation. Specifically, we have developed and evaluated the following:

1. Control-C, a dialect of C, which is expressive enough for control and embedded systems and can be analyzed to guarantee memory safety without run-time software checks or garbage collection.

   (a) We developed a novel analysis that ensures that dangling pointer dereferences are memory safe. In this context, we developed an

analysis method to determine programs that potentially have increased memory consumption.

(b) Our experiments showed that our analyses proved memory safety without run-time software checks or garbage collection for a large class of control and embedded systems. (This class is even larger if run-time checks are inserted for bounds-checking certain array indexing in the program).

2. Safe value propagation:

   (a) We have identified multiple implementation errors that can violate value propagation paths between non-core and core components. We statically verify that critical components *monitor* values from unreliable components before using them. We detected multiple hidden dependencies in the simplex implementations for different control systems.

   (b) We have developed a simple annotation language that can represent semantic information regarding run-time monitors and critical information in the core component.

   (c) Our analysis forms the verifiable "last line of defense" against the effects of many complex, difficult-to-detect errors including data races and data incompatibilities.

3. A co-design based architecture to make networked control systems tolerant to timing errors such as delays and restarts of individual components.

Our approach has a few limitations that need to be addressed in future work. First, we handle only single-threaded programs. Secondly, our analysis to support SP2 (temporal isolation) simply involves restricting the component to invoke a subset of system calls assuming the presence of a run-time system to regulate system resources granted to the non-core subsystem. In the future, the requirement imposed on the run-time system can be reduced with the aid of additional static analyses. Finally, particularly in detecting the set of values in the core component affected by an unmonitored, unsafe non-core value, static analyses can result in false positives along longer paths. In the future, ranking of these results can aid the manual inspection of the potential erroneous dependencies reported.

1.6 Prior Work on Robust System Development

Prior work on robust system development has focused on two complementary approaches – robust system architectures and tools that enforce archi-

tectural properties. The respective utility of architectural and tool-based approaches varies depending on their precision, scalability, and their position in the design-develop-test-execute cycle. In particular, formal verification has been applied at design time; static analysis and testing at development time, and run-time monitoring and dynamic analysis at execution time.

1.6.1 Architectural Solutions for Robustness

Traditionally, fault tolerance in dependable systems has been achieved through replication [85]. While replication is very successful technique in hardware which tend to have random errors errors, it has proved ineffective in software, except for race conditions, which are indeed random. Algorithmic errors and other software bugs when replicated produce the same erroneous values across all the replicas. In addition, as mentioned in the previous section, the upgrade paradox shows that an upgrade is ineffective when effected in a minority of the replicas and is vulnerable to bugs in the upgrade when effected in a majority of the replicas. Finally, redundancy through n-version programming [7] has proved expensive in practice and studies [7] have shown that different teams of programmers tend to make similar errors while developing complex systems.

Rushby [79] and ARINC Specification 653 [23] describe a comprehensive set of requirements for partitioning components in IMA Firstly, Rushby describes the "alternative gold standard" to be that the behavior and performance of software in one partition must be unaffected by the software in other partitions. ARINC Spec. 653 describes *spatial partitioning* as the property that the software in one partition cannot change the software or private data of another partition or access private actuators and devices. Enforcing this requirement requires protection against memory errors and system resource errors that hog resources that have been dedicated to another component. *Temporal partitioning* ensures that the service received from shared resources by software in one partition cannot be affected by the software in another partition. Enforcing this requirement prevents the errors due to corruption or hogging of shared system resources including timing issues due to poor scheduling of the CPU. The alternative gold standard does not sufficiently address the value propagation errors between components in different partitions due to communication.

Rushby [79] describes two designs for partitioning components: application-level partitioning on a common operating system and a virtual machine architecture, which provides separate operating system services for each partition above a minimal kernel. In highly critical systems like avionics, the virtual machine architecture, which sandboxes resources, devices, and software components in a single partition thus providing a virtual ma-

chine for each partition, is an attractive choice. Verifying this architecture involves formally guaranteeing that each partition is spatially and temporally separated with respect to all hardware registers, resources, devices, and address space. The virtual machine architecture is too heavy-weight for the purposes of general purpose control and other embedded systems.

For general purpose embedded systems, the feature separation between core and non-core functionality provided by Simplex [82] constitutes the state of the art in robust architectural design. Fault-tolerant system design properties such as *masking fault-tolerance* also rely on separation between critical and non-critical functionality [5].

In addition, there are several miscellaneous techniques that provide robustness against value and delays in control systems. Cunha *et al* [53, 26] have described the mechanism of using a fail-bounded model as a run-time monitor to protect against controller value errors due to transient errors. The monitor verifies that the controller output is within the bounds of stability, taking recovery measures if a controller may bring down the system. State estimators like the Kalman filter [21] have been used in control systems to protect against sensor value errors. Also, there is a wealth of control theoretic literature as listed in [75], which describe estimation filter techniques for controllers in the presence of communication and sensor delays.

1.6.2 Tools for Robustness

A diverse set of tools have been developed in literature for use at different points of the development cycle. Domain experts use modeling tools such as Matlab and Simulink, which are useful in developing the set of the equations that govern the physical world. In very simple embedded systems, this prototype can be used to generate code (or code stubs) itself. More complex architectures satisfy properties which can be verified using architecture description languages. Apart from SAE-AADL [77], which was described earlier, fault tree analysis [24] is an extensively used design tool used to describe the propagation of faults across components. However, in most cases fault trees are generated manually by an expert. Leveson *et al* [69] have developed an approach to generate fault trees from source code using a manual technique. The fine-grainedness of the faults and the manual approach results in very large fault trees in practice. Arora and Kulkarni [5] have demonstrated the design of masking fault-tolerant systems, whose architecture they formally analyze. Their analysis is carried out on an abstraction (albeit, detailed), which is different from the actual implementation.

Several static analysis tools have been developed to verify global robustness properties. SFI [91], CCured [22], Cyclone [55], and Java [45, 12] have either prevented or sandboxed memory errors through run-time checking,

compiler-based static analyses or language mechanisms, or through a combination of these techniques. However, all of them are unsuitable for embedded systems due to their run-time overheads, platform dependence and porting effort. (This is further discussed in chapter 2). Giotto [51] is a language that enables specifying the timing requirements of hard real-time components, whose schedulability can be verified by the compiler. Thus timing requirements of unreliable components in the context of the remainder of the system can be verified before these components are executed. The Ada-SPARK system [8] defines a safe subset of the Ada language which statically prevents memory errors and utilizes annotations to make the data flow information in a program explicit, thus avoiding implicit value propagation errors for a subset of Ada.

Run-time monitoring [49, 18, 58] verifies that assertions and invariants in the code hold at execution time. It is useful for consistency properties and for error handling within a single component at execution time. These invariants or assertions are most useful in analyzing functional properties. Testing based techniques to determine failure dependency graphs such as automatic failure-path inference [15] are useful analytical tools, but do not provide any guarantees. Moreover, testing an integrated system is less and less tractable as systems become more complex [86].

1.7 Research-Scale Control Systems

In order to experimentally evaluate our techniques for robustness, we use three laboratory scale projects that are complex enough to demonstrate our ideas. Independent sources from the industries such as Lockheed Martin have observed that these research-scale systems are representative of real systems in the field. In both the systems described below, a system integration architect developed a robust architecture, while 1-3 domain experts and software developers developed the domain-specific equations and the individual components respectively.

The inverted pendulum shown in figure 1.1 is a classic real-time feedback control system which is balanced by adjusting the position of the cart. The control periodically receives track position and pendulum angle as feedback from the sensor. The Simplex architecture is used in this system to enable online upgrades of high-performance controllers, while keeping the pendulum upright even in the presence of bugs or malicious attacks in an unreliable controller.

The traffic control testbed was developed in the IT Convergence Lab at CSL, UIUC, as a research prototype to study networked control systems [46]. The testbed, shown in figure 1.2, consists of autonomous cars operating on an indoor track in various traffic scenarios. The trajectory of a

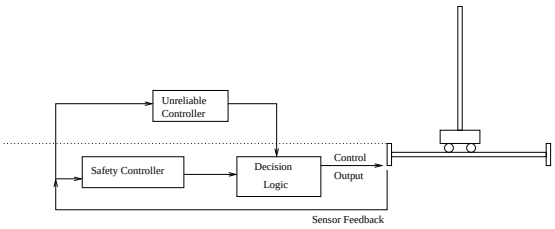


Figure 1.1: Simplex for the Inverted Pendulum

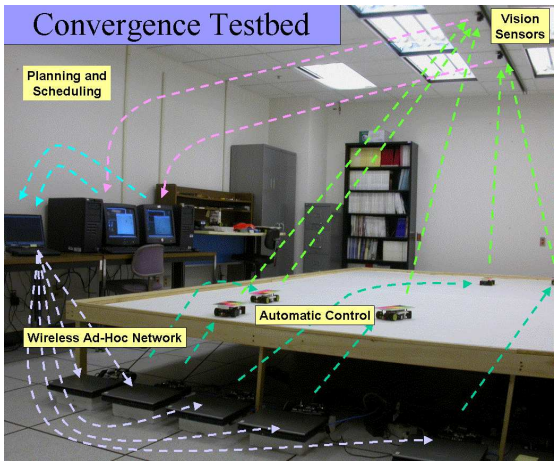


Figure 1.2: Traffic control testbed

car is determined by its speed and orientation, and control is accomplished by changing these parameters. Currently, cars have no on-board computing capability, and are controlled through radio frequency transmitters on dedicated channels. Control for each car is computed by software executing on a corresponding laptop. For our purposes, however, this configuration is equivalent to each car having a dedicated on-board computer.

Cars are tracked by a vision system consisting of two ceiling mounted cameras. The video stream from each camera is processed by software executing on a corresponding desktop computer. This software scans a video stream as a sequence of frames, and extracts positions and orientations of cars in the field of vision. This information is then available as feedback to control the cars.

1.8 Organization of Chapters

The remainder of this report is organized as follows. Chapter 2 describes the background work on compiler and run-time techniques to guarantee memory safety for programs. We also describe previously developed foundational compiler analyses on which we have built our own analyses. It then goes on to describe Control-C, a dialect of C, which is expressive enough

for a broad class of embedded and control systems, which enables us to guarantee memory safety with no run-time overhead. Chapter 3 discusses the notion of *use* and *depend* relationships between communicating components and provides an accompanying compiler analysis to enforce that critical components only use data from unreliable components without depending on it. Chapter 4 synthesizes the techniques in the earlier chapters in the context of building a robust system. It also describes mechanisms to extend our techniques to a broader class of applications. We conclude with chapter 5.

Chapter 2

Spatial Isolation

Programming environments, including programming languages, compilers and run-time systems, play a vital role in improving the reliability and efficiency of software. A key class of properties provided by these environments is that of program safety, i.e., prevention of semantic errors such as type violations or memory access violations will not be allowed to occur undetected. Such safety guarantees provide three benefits. First and foremost, they enable development of reliable software by detecting and preventing programming errors at compile time or run time. Second, they ensure better isolation between different components or modules of a software system, which is particularly important for safety-critical software. Finally, static analysis enables checking many errors at compile time. Early detection of errors in the development cycle instead of run-time detection is important for many critical systems with strict performance and energy constraints.

Embedded systems and control systems today avail of practically none of the benefits offered by modern programming environments. They are often written in type-unsafe languages like C and C++, which do not prevent a host of errors including uninitialized variables, dangling pointers, and buffer overflows. In this chapter we first discuss the importance of memory safety for embedded system components, define our goals, and finally outline our approach and our prior work on which we have built our solution. Our approach and the comparison with other related work is discussed in detail in the following chapter (section 2.4).

2.1 Motivation, Definitions, and Goals

Memory errors in buggy components can overwrite and corrupt memory locations allocated to other components in the system, including critical components, system memory, and other reliable infrastructure. This problem is particularly relevant to embedded systems, where components are not protected by address space boundaries such as process boundaries in Unix. This has proved to be an important problem in practice as evidenced by the Phobos I disaster [19], where a keyboard buffer overflow corrupted a critical aircraft control function. The Simplex architecture [82, 81] protects

critical components against value, timing and run-time faults in unreliable components, but does not prevent memory errors. Simplex also uses the process abstractions to sandbox some components, but such protection may be unavailable. Moreover, processes do not prevent violation due to malicious memory errors. Thus, the infrastructure is vulnerable to buggy and malicious codes, which can potentially corrupt critical components.

Another manifestation of memory errors is the execution of illegal system calls from the data area of a component, through simulated buffer overflows and other memory errors. Three such attacks (there are many similar ones) are illustrated in figure 2.1, 2.2, and 2.3. The code snippets shown illustrate the mechanisms that can be used as malicious attacks through memory errors on Linux 2.2.18. In each of the attacks, the `killcode` array shown in Fig. 2.1 represents binary code for the StrongARM that executes the `kill(-1, 9)` system call, so that the code is hidden in the data area. Each attack uses a different mechanism to jump to the array address in order to execute this code. Figure 2.2 uses a cast from a character string to a function pointer is used to jump to the code in the `killcode` array. Fig. 2.3 demonstrates the use of uninitialized pointers to compromise the system. Invocation of the function `init` initializes the stack with values such that when `func` is invoked, the local pointer variable `p` in `func` contains the address of the return address of `func` on the stack. Dereferencing `p` modifies the return address. This in turn, results in the the skipping of the instruction decrementing 'i' after the return, leading to the overflow of `mainbuf`. This in turn overwrites the return address of `controlFunction` with the address of `killcode`.

```
char killcode[] =
    "\x55\x89\xe5\x89\xe5\xb9\x09\x00\x00
     \x00\xbb\xff\xff\xff\xff\xb8\x25\x00
     \x00\x00\xcd\x80\x89\xd3\xc3\x90";
```

Figure 2.1: Source for the `killcode` array

```
void controlFunction(float a, float b) {
   void (*func)();
   func = (void (*)) &killcode[0];
   func();   // jump to &killcode[0]
   ...
}
```

Figure 2.2: Illegal cast

Motivated by the above effects of memory errors, we define memory safety as follows:
A software entity (a module, thread or complete program) is *memory safe* if (a) it never references a memory location outside the address space allocated by or for that entity, and (b) it never executes instructions outside the

```
void func() {
  int *p;     /* contains address of return
                 address stack slot */
  (*p) += 22; /* modifies return address */
}
void init() {
  long i;
  i = &i - 2; /* Store address of return
}              * address stack slot */
void controlFunction(float a, float b) {
   int mainbuf[3], i = 4;
   init();      /* Stack initialized */
   func();             /* returns 3 lines down */
   if ((int)a % 2) {
     i -= 3;                /* never executed */
     mainbuf[i] = killcode;  /* force return to
                                &killcode[0] */
   }
}
```

Figure 2.3: Illegal return via an uninitialized variable

code area created by the compiler and linker within the address space.

This definition of memory safety incorporates spatial separation (SP1) from chapter 1. In addition, it also guarantees that there are no hidden system calls that can be executed from its data area, which, in turn, enables verification that the set of system calls belong to a safe permitted subset. This property is a prerequisite to guaranteeing temporal isolation (SP2) and safe inter-component interaction (SP3).

Our goal in this work is to enforce memory safety for embedded systems written in languages like C and C++ with no run-time software checks or garbage collection. In particular, we define a subset of the C language, which is restrictive enough to guarantee memory safety through static analysis alone, but is expressive enough to program a broad class of embedded and control systems. We ensure that there are not syntactic changes and only semantic restrictions to avoid any porting effort.

2.2 Memory Safety: The SAFECode Approach

Our approach to guarantee memory safety is a part of the SAFECode project (see
`http://safecode.cs.uiuc.edu`) in collaboration with Dinakar Dhurjati at UIUC. In particular, the contributions of this chapter and Dhurjati's dissertation [30] provide a complete solution to the memory safety problem. Dhurjati's contributions will be distinctly highlighted at the appropriate points.

The SAFECode technique consists of identifying all causes of violation

of memory safety in type-unsafe languages such as C and consequently performing a series of compiler analyses that enforce that these violations are detected and prevented or that they do not cause memory safety violations. Significantly, we apply only minor semantic restrictions on C programs, while still obtaining all the benefits of safe languages, without using garbage collection or run-time software checks. The minimum guarantee provided is memory safety as defined in the previous section. In addition we provide significant error detection, similar to strongly type safe languages, with one exception: we allow dereferencing of dangling pointers but enforce that such dereferences are, in fact, safe. The latter clause violates the technical definition of strong type safety. Furthermore, we rely on some assumptions on the hardware-supported address space protection (which are reasonably commonplace).

2.3 Prior Work

Our compiler analyses for memory safety take advantage of two existing analyses, the data structure analysis [62] and automatic pool allocation transformation [64], in addition to a previous established key principle for memory safety, the *type homogeneity principle*, used to guarantee safety of dangling pointer dereferences. The two analyses and the principle are described in this section.

2.3.1 Data Structure Analysis

Data Structure Analysis (DSA) is a pointer analysis algorithm that is carefully designed to identify disjoint instances of entire pointer-based data structures and their lifetimes, while remaining fast and scalable enough for large, realistic programs. DSA computes a points-to graph representation we call a Data Structure Graph (DS Graph). DS graphs provide all of the information used in the rest of this work (including Automatic Pool Allocation). We describe DS graphs first and then briefly discuss the properties of DSA needed to achieve our goals while keeping it efficient and scalable.

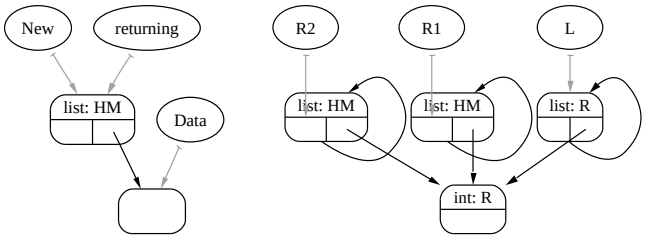
A DS Graph captures compile-time information about the memory objects created by a program and the pointer relationships between them. A separate DS graph is computed for each function in a program, except that all functions within a strongly connected component of the call graph share a single, common points-to graph (we do not try to be context-sensitive within such recursive regions). Different nodes within the same graph represent distinct memory objects. Formally, a **DS Graph** is a directed multi-graph, where the nodes and edges are defined as follows:

DS Node : A DS node is a 5-tuple $\{\tau, F, M, A, G\}$. τ is some program-defined type, or $\perp$ representing an unknown type. In the analysis,

```
struct list { list *Next; int *Data; };
list* createnode(int *Data) {
  list *New = malloc(sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(list *L, list **R1, list **R2) {

  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(L->Data);
    splitclone(L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(L->Data);
    splitclone(L->Next, R1, &(*R2)->Next);
  }}
```

(a) Fragment of C program manipulating linked lists



(b) DS Graphs for `createnode()` (left) and `splitclone()`.

Figure 2.4: Example illustrating Data Structure Graphs

$\perp$ is treated like an unknown-size array of bytes. F is an array of fields, one for each possible field of the type τ. Scalar or array types have a single field. M is a set of memory classes, written as a subset of $\{\mathbf{H}, \mathbf{S}, \mathbf{G}, \mathbf{U}\}$, indicating Heap, Stack, Global and Unknown memory objects respectively. A **U** node is assigned type $\perp$. Finally, if $\mathbf{G} \in M$, then G is a non-empty set of global variables and functions included in the objects for this node; otherwise, G is empty. Finally, A is a boolean that is true if the node includes an Array object.

DS Edge : A DS edge is a 4-tuple: $\{s, f_s, t, f_t\}$, where s and t are DS nodes, and f_s and f_t are fields of s and t respectively. Thus, the graph provides a field-sensitive representation of points-to information. A field of a node may have no outgoing DS edge only if the field is known not to contain a pointer type, e.g., it is a function, floating point, or small integer type, or if $M = \{\mathbf{U}\}$.

Figure 2.3.1(b) shows the DS graph computed for function `splitclone` of an example program, computed by our compiler. Note that each node of

type `list` has two fields. The cycles indicate recursive data structures.

The DSA algorithm, which computes DS graphs, is described in [62]. DSA is "fully context-sensitive" in the sense that it names heap objects by entire acyclic call paths (which we refer to as "full heap cloning"), and it is field-sensitive, i.e., it distinguishes distinct pointer fields within a structure. Being fully context-sensitive is important because it allows the analysis to distinguish heap objects that may be created, processed, and destroyed by calling common functions. This enables Automatic Pool Allocation (which is based on DS graphs, as described below) to put distinct instances of the same logical data structure into distinct pools in many cases. In the DS graph for `splitclone` in Figure 2.3.1, $R1$ and $R2$ point to distinct nodes, indicating that the analysis has proved the two linked lists are completely disjoint. This allows pool allocation to put these two lists in distinct pools, even though they are created in an interleaved fashion and by calling the same function.

To achieve speed and scalability, DSA is flow-insensitive and it uses a unification-style analysis, i.e., a pointer field at a node has exactly one outgoing DS edge so that all pointer target objects of the pointer are merged into a single graph node. We argue in [62] that the combination of full heap cloning with unification is scalable to very large programs in practice. DSA correctly analyzes non-type-safe programs and incomplete programs, and infers the call graph incrementally as part of the analysis using a new, non-iterative technique. The first two properties are essential for real-world use.

DSA actually computes multiple DS graphs for each function. The "complete bottom-up" DS graph for a function incorporates the effects of all functions reachable from the current function (i.e., immediate callees and their callees and so on), including functions called via function pointers [64]. The final, "top-down" DS graph of a function incorporates the effects of both the callers as well as the callees of a function, so that it captures the full set of memory objects and aliasing relationships from all possible call-sites (as well as those due to side effects of callee functions).

We have evaluated DSA experimentally for over 35 C programs, and found that it is both extremely efficient and scales well across a large range of program sizes [62]. DSA requires about 2-8 seconds and less than 16MB of memory for several C programs ranging in size from 60K to 130K lines of code. Empirically, it also scales almost linearly in analysis time for 35 benchmarks spanning 4 orders-of-magnitude of code size. No previous algorithm we know of has demonstrated both speed and scalability with full heap cloning. DSA is compared with previous pointer analyses in more detail in [62].

2.3.2 Automatic Pool Allocation

Given an ordinary imperative program that uses explicit allocation (e.g., `malloc`) and deallocation (e.g., `free`), the Automatic Pool Allocation transformation [64] rewrites the program to segregate heap objects into multiple pools, by performing allocation and deallocation operations from those pools. The transformation attempts to use separate pools of memory for each logical data structure instance (e.g., a particular linked list or a graph) that is not exposed to unknown external functions. This differs from other automatic region inference algorithms that infer regions primarily for performing automatic memory management [88, 20], which do not consider data structure relationships in choosing how to partition objects into regions.

We use a pool allocation library with five simple operations:

(a) `poolinit(Pool** PP, unsigned size)` allocates and initializes a new pool descriptor for objects of the specified size;

(b) `pooldestroy(Pool* PP)` clears the pool descriptor and releases the remaining memory in the pool back to the system heap;

(c) `poolalloc (Pool* PP, unsigned nbytes)` allocate a single object or an array of objects in the pool, depending on `nbytes` and the size of the objects in the pool and

(d) `poolfree (Pool* PP, T* ptr)` deallocates an object within the pool by marking its memory as available for reallocation by `poolalloc`.

A pool is created before the first allocation for its data structure instance and destroyed at a point where there are no accessible references to data in the pool. The pool library internally uses ordinary `malloc` and `free` to obtain memory from the system heap and return it when part of a pool becomes unused or the pool is destroyed.

To illustrate the pool allocation transformation we use the example in Figure 2.5. In this example, function `f` calls `g`, which first creates a linked list of 10 nodes, initializes them, and then calls `h` to do some computation. `g` then frees all of the nodes except the head and then returns.

The pool allocation transformation operates as follows:

1. *Identify pools within each data structure*: We traverse the complete bottom-up Data Structure Graph (DSG) of each function to identify heap nodes. Each heap node in this DSG corresponds to objects of a single data type, allocated within the current function or one of its callees. Objects corresponding to this node are allocated in a single pool.

```
f() {
  ...
  g(p);
  // p->next is dangling
  p->next->val = ... ;
}

g(struct s *p) {
  create_10_Node_List(p);
  initialize(p);
  h(p);
  free_all_but_head(p);
}

h(struct s *p) {
  for (j=0; j < 100000; j++) {
    tmp = (struct s*) malloc(sizeof(struct s));
    insert_tmp_to_list(p,tmp);
    q = remove_least_useful_member(p);
    free(q);
  }
}
```

Figure 2.5: Pointer safety and pool allocation example

2. *Identify where to create/destroy pools*: For each procedure, the DSG can be used to identify those DS nodes that are not accessible after the procedure returns (i.e., nodes that are not reachable from globals, formal arguments and return value). For each such node, we insert calls to create and destroy the corresponding pools of memory at the entry and exit of the procedure.[1] In our running example, the linked list does not escape from the procedure `f()` to its callers and so we create and destroy the pool for the list in procedure `f()`, as shown in Figure 2.6.

3. *Transform (de)allocation operations and function interfaces*: We transform all `malloc` and `free` calls in the original program to use the pool allocation versions, as illustrated in function `h()`. For any function containing such operations on a pool created outside the function, we add extra arguments to pass the appropriate pool pointers into the function (and do the same for possible callers of such functions, and their callers and so on). The transformation uses the call graph constructed by DSA for all interprocedural steps, and correctly handles programs with function pointers and recursion. The changes are illustrated by the functions `g()` and `h()` and their invocations in Figure 2.6.

[1] Our pools do not require nested lifetimes. We could move `poolinit` later in the function and move the `pooldestroy` earlier or into a callee using additional flow analysis, but we do not do so currently.

```
f() {
  Pool *PP;
  poolinit(PP, sizeof(struct s));
  ...
  g(p, PP);
  // p->next is dangling
  p->next->val = ... ;
  pooldestroy(PP);
}

g(struct s *p, Pool *PP) {
  create_10_Node_List(p, PP);
  initialize(p);
  h(p, PP);
  free_all_but_head(p, PP);
}


h(struct s *p, Pool *PP ) {
  for (j=0; j < 100000; j++) {
    tmp = poolalloc(PP);
    insert_tmp_to_list(p, tmp);
    q = remove_least_useful_member(p);
    poolfree(PP, q);
  }
}
```

Figure 2.6: Example after pool allocation transformation

The result of this transformation for type-safe programs is that all heap-allocated objects are assigned to type-homogeneous pools, nodes in disjoint data structure instances identified by DSA are assigned to distinct sets of pools, and individual items are allocated and freed from the individual pools at the same points that they were before. A pool is destroyed when there are no more live (i.e., reachable) references to the data in the pool.

Note that the transformation as described so far *does not ensure program safety*. Explicit deallocation via `poolfree` can return freed memory to its pool and then back to the system, which can then allocate it to a different pool. Dangling pointers to the freed memory could allow data of arbitrary types to be accessed, and could violate memory safety.

2.3.3 Type Homogeneity Principle

The primary challenge in enforcing memory safety without garbage collection and with explicit deallocation is to detect dangling pointer dereferences or ensure memory safety in the presence of such dereferences. We adopt the latter approach through the use of the following principle:
(Type homogeneity principle) If a freed memory block holding a single object were to be reallocated to another object of the same type and alignment, then dereferencing dangling pointers to the previous freed object cannot cause a

type violation.
This principle implies that to guarantee memory safety, we do not need to prevent dangling pointers or their usage in the source — we only need to ensure that they cannot be dereferenced in a type unsafe manner. The principle allows correct programs (i.e., programs with no uses of dangling pointers) to work correctly without any runtime overhead. Programs with dangling pointer errors will execute safely but we will not detect such errors for these programs.

It is important to note that this principle only enables dangling pointers to dereference memory within the same thread. Also, pools only contain memory within a single thread and reallocate memory only to the same thread. Therefore, in the event of a dangling pointer dereference in the core component, the dangling pointer can only access memory within its own thread and cannot inadvertently access data in a non-core component. This is vital to the property verified in the following chapter (chapter 3): explicit communication between core and non-core components (threads).

One objection to achieving safety via the above principle is that it can make it *more difficult* for programmers to detect dangling pointer errors because such errors would not produce any type violations. During debugging, however, the above principle *need not be used*. In fact, the pool allocation library and runtime system can use many run-time techniques to assist in detecting dangling pointer errors. During production runs, on the other hand, we believe the principle *is* appropriate to use and its benefits greatly outweigh any possible loss in error detection. During production runs of embedded software or system software, there is little benefit in halting execution as soon as a potential memory corruption occurs (in fact, many memory corruption errors may not lead to significant failures, so halting execution immediately could be premature). This has been demonstrated and exploited in work on failure-oblivious computing [78]. The benefits of ensuring memory safety despite such errors (while avoiding the overheads of garbage collection) are much more significant for such software.

Type Homogeneity Principle for Heap Safety

Using the above principle directly, one naïve but impractical *and incorrect* solution is to separate the heap into disjoint pools for distinct data types and never allow memory used for one pool to be reused later for a different pool. This is impractical because it can lead to large increases in the instantaneous memory consumption. The worst-case increase for a program with N pools would be roughly a factor of $N - 1$, when a program first allocates data of type 1, frees all of it, then allocates data of type 2, frees all of it, and so on. More importantly, the simple solution is incorrect because it would allow errors that make the results of the compiler's pointer analysis invalid,

and therefore invalidate any static checks that use pointer analysis (including our stack safety, and array bounds checking algorithms). This is briefly discussed below and is the main subject of Dhurjati's dissertation [30].

An technique that partitions the heap into type homogenous memory partitions should enforce that a dangling pointer can only refer to objects from the same pool as the original object. In addition, a dangling pointer can only refer to objects pointed by pointers that are aliased to the dangling pointer according to the pointer analysis (such as DSA). Violating the latter property can invalidate the results of the pointer analysis. Consider the following sequence: `free(q); ...; p = malloc(...)`. If `q` is still usable (i.e., it is a dangling pointer) and `p` has reused the memory freed by `q` since they point to the same pool, then it is essential that `p` and `q` are aliased according to the pointer analysis. In other words, the pointer analysis can be no more precise than the segregation of objects into pools. Pointer analysis correctness is essential since other analyses such as stack safety analysis and array bounds analysis for safety use the pointer analysis for alias information.

Note that the naïve solution of using one pool per static type would *not* ensure correctness of pointer analysis. This is because, in the above example, `p` could reuse the memory of `q` even though `p` and `q` are unaliased according to the pointer analysis. Thus, `*p` and `*q` would be aliased in this execution even though pointer analysis claimed they were not.

Our solution is essentially a more sophisticated application of this basic principle, using Automatic Pool Allocation to achieve type-homogeneous pools with much shorter lifetimes in order to avoid significant memory increases as far as possible. Since our pools are already type-homogeneous, we simply need to ensure that the memory within some pool P_1 is not used for any other dynamically allocated data (either another pool P_2 or heap allocations within trusted libraries) until P_1 is destroyed. This can be done easily by modifying the runtime library so that memory of a pool is not released to the system heap except by `pooldestroy`. With this change, any reference via a dangling pointer to a pool object will be guaranteed to reference either the original object or a new object of the same type and alignment as the original, and *belonging to the same pool*. This ensures that the basic principle described above is satisfied.

Note that the approach preserves the semantics of the pointer analysis, viz. DSA (This is treated formally in [30]). Informally, a dangling pointer through the restriction above can only point to freed memory corresponding to the same type and with the same alignment in the type-homogenous pool corresponding to its DS Node. This enforces that it can be aliased to other pointers pointing to this DS Node, which is exactly equivalent to the results of the DSA pointer analysis. Note that our solution based on Automatic Pool Allocation ensures that the results of DSA are correct but if some static

analysis (e.g., array bounds checking) used a more precise pointer analysis than DSA, the results of such a pointer analysis may be illegal for some executions due to dangling pointer references.

2.4 Control-C: Memory Safety Without Run-time Overhead or Garbage Collection

The goal of this part of our work is to provide a programming environment for embedded system, which guarantees memory safety without run-time software checks or garbage collection. To this end, we develop a language, Control-C, which is C with some semantic restrictions that enable guaranteeing memory safety through static analysis without run-time overhead or garbage collection, but still makes it expressive enough to program a broad class of embedded and control applications. In particular, the language should be able to utilize complex pointer based data structures and permit explicit deallocation. Also, we do not want introduce any programmer annotations, in order to eliminate any porting effort for legacy codes. To achieve this goal, we first examine the causes of memory safety violation in type-unsafe languages:

1. *Bad casts*: When a memory location is used as two or more different types due to casting from one type to another.
2. *Uninitialized pointers*
3. *Dangling pointers to freed memory*
4. *Array bounds violation*: Indexing the array beyond its size causes memory safety violation. Pointer arithmetic and array accesses are treated uniformly in C.
5. *Dangling pointers to the stack*: A pointer to a stack location allocated within a function, after the function returns.

Further, we first discuss the semantic restrictions imposed in Control-C. These restrictions are basic restrictions that enforce type-safety. We then discuss analyses to address the memory safety implications of each of the above memory errors. An early version of Control-C was highly restrictive, but was still useful in programming control applications. This described in our paper [60] and has been documented in appendix A. The work described in this chapter is necessarily a superset of (and is far less restrictive than) the initial version of Control-C and has been published in [32] and [33].

Memory safety without run-time overhead is an extremely difficult, if not impossible, goal for type-unsafe languages. As a result we impose certain semantic restrictions on these languages. Note that we avoid any new language mechanisms or syntactic changes, in order to avoid significant porting effort. First of all, we restrict ourselves to programs that follow basic type-safety rules. Second, some languages constructs such as unanalyzable array references and pointers to stack-allocated memory that outlive stack frames, make it impossible to ensure memory safety without run-time software checks. We reject programs that have these constructs. This approach of imposing semantic or usage restrictions on a programming language for the purposes of analyzability has been adopted in other attempts at defining languages for safety-critical systems such as SPARK-Ada [8]. Failure to comply with our restrictions can necessitate run-time software checks for programs that do not adhere to our restrictions or program restructuring to adhere to Control-C language rules. In this work, we do not insert any run-time checks wherever necessary and instead simply reject input programs that require them. Although, extensions to this work by Dhurjati [30] can handle all kinds of type-unsafe constructs in C or C++. This has been described in section 2.10. Finally, our semantic restrictions have been defined in a language-independent way on the low-level virtual machine (LLVM) instruction set [63]. Thus our restrictions and the accompanying analyses which operate on LLVM can be used for programs in any source language by compiling it to LLVM.

2.5 Assumptions in Control-C

Achieving memory safety and preventing memory access errors under the constraint that we permit explicit memory deallocation instead of requiring an automatic memory management and that we avoid any run-time software checks before program operations, is a challenging one. We require some run-time support and certain system assumptions, which we consider acceptable for a majority of embedded platforms (hardware + OS). These assumptions are summarized below.

First, we make some assumptions about the runtime environment. We assume that certain run-time errors are safe, i.e., the runtime system can recover from such errors by killing the applet, thread, or process executing the untrusted code.

We assume a safe run-time error is generated if either the stack or the heap grows beyond the available address space.

We assume the system has a *reserved address range* and any access to these addresses causes a safe run-time error, typically triggered by a page fault handler or by a reserved address range in hardware on systems with-

out virtual memory management. In practice, embedded platforms vary widely in their addressability.

- In standard Linux implementations with 32-bit addressing, the high end of the address space is reserved for the kernel (typically 1GB out of 4GB). Our technique to handle null pointer dereferences described in Section 2.7.1 is most suited to such platforms, where we avoid run-time software checks.
- Smaller embedded platforms with 16-bit and even 8-bit addressing typically do not have a reserved address range. In these cases, null pointer checks must be inserted before every load or store.

Rule (P2) in section 2.7.1 requires that the size of any structure not exceed the size of the reserved address range. In the event that a program contains a structure larger than the size of the reserved address space, the programmer can either restructure the code to use smaller structures or run-time null pointer checks are necessary for any references to the structure.

We assume that certain standard library functions and system calls are trusted and can be safely invoked by the untrusted code (calls whose arguments must be checked are discussed in Section 2.8). We assume (and check) that the source code of all other functions is available to the compiler. We also require that the program be single-threaded.

2.6 Basic Language Restrictions

The first set of restrictions are the typing rules that are summarized below. We assume a low-level type system including a set of primitive integer and floating point types, arrays, pointers, user-defined records (structures), restricted union types, and functions.

(**T1**) Our type system is the same as that of the C language, but is further restricted by rules **T2** and **T3**.

(**T2**) Casts *to* a pointer type from any other type are disallowed, except certain pointer-to-pointer casts for compatible targets. Permitted casts include casts between pointers to primitive types of the same size or casts from a pointer to a primitive type to a pointer to a primitive type with a smaller size.

(**T3**) A union can contain only types that can be cast to each other, e.g., a union cannot include a pointer and a non-pointer type.

Rule (**T3**) is similar to Rule (**T2**) as unions are implemented using casts. The exceptions to Rule (**T2**) are essentially reinterpreting casts for the target numerical value. Note, however, that if the pointer points to an array,

the resulting pointer would have no size information and hence any subsequent array index operations would likely be rejected as unsafe by the array bounds checking algorithm (Section 2.8). Explicit array declarations are just as in C, i.e., they need to specify the size of each dimension, except for the first dimension of an array formal parameter. Enforcing the above rules is trivial in LLVM [63], where all operations are typed and only an explicit `cast` instruction can be used to perform any type conversion.

2.7 Safety of Pointer References

Memory safety of pointer references is violated through uninitialized pointers, dangling pointers to the stack and dangling pointers to freed heap memory.

2.7.1 Uninitialized Pointers

We employ two restrictions, (P1) and (P2) below, in addition to our requirement that the runtime system have a reserved address range, to ensure that an uninitialized pointer (scalar or an element of an aggregate object) is either never dereferenced or results in a safe run-time error.

(**P1**) Every local pointer variable must be initialized before being referenced, i.e., *before being used or having its address taken*.

(**P2**) Any individual data type (i.e., not an array) should be no larger than the size of the reserved address range.

Our compiler prevents errors due to uninitialized pointer values by statically checking that the program honors rules (**P1**) and (**P2**). Rule (**P1**) is motivated by the following code snippet, disallowed by our language:

```
int a, *p, **pp; pp = &p; print(**pp); p = &a;
```

Here, the address of uninitialized pointer `p` is taken before it is initialized, thus making `**pp` potentially unsafe. Such uses are difficult to detect statically (because the use may be in a different function), and even a flow-sensitive interprocedural algorithm is likely to lead to false errors. We prefer to disallow taking the address of an uninitialized pointer. We use a standard global data flow analysis to check rule (**P1**) above, that considers only local scalar pointer variables. (Note that interprocedural analysis is not required for identifying uninitialized variables, since any variable needs to be initialized in the calling function before it is passed as an argument).

Detecting uses of uninitialized values for global variables and for pointers within dynamically allocated data (e.g., structure fields or array elements) is difficult at compile time. Type-safe language implementations

usually initialize pointer fields in aggregate objects to null and use run-time null pointer checks to detect uses of uninitialized values. In order to avoid performing such checks explicitly in software, we initialize all uninitialized global scalar pointers, and all pointer fields in globals and dynamically allocated data structures at allocation time, to point to the base of the reserved address range. This is enabled by our typing rules, which ensure that the type of each dynamically allocated object is known statically. Pointer fields in stack-allocated variables of aggregate types are also initialized to the same value. This includes arrays of pointers in the aggregate type which are initialized in a loop only once at allocation time. Finally, the constant 0 used in any pointer-type expression is replaced with the same value. Rule (P2) above specifies that the size of any individual structure type[2] cannot exceed the size of the reserved address range. With this rule, the above initialization ensures that the effective address for the load or store of any scalar variable or structure field using an uninitialized pointer (e.g., `p->X`, where `p` is uninitialized) will fall within the reserved address range, thus triggering a safe run-time error. If a reserved address range is unavailable or the structure size restriction above is unacceptable, then explicit software checks for null pointer references would be required.

2.7.2 Stack Safety

The second way in which pointer usage can lead to unsafe memory behavior (problem (b) in section 2.6) is when a pointer into a stack frame of a function is live after the lifetime of the function. This potentially arises when the address of a local variable is made accessible after the function returns. To avoid this problem, many type-safe languages like Java disallow taking the address of local variables. We choose to be less restrictive: we disallow only placing the address of a stack location in any heap location or global variable, or returning it directly from a function (rule **P3** below). Microsoft CLR's type system system [44] has exactly this restriction.

(**P3**) The address of a stack location cannot be stored in a heap-allocated object or a global variable, and cannot be returned from a function.

Rule **P3** can be enforced using a simple traversal of the Data Structure Graph for each function, checking whether any stack-allocated object is reachable from the function's pointer arguments, return node or globals. Note that this is equivalent to traditional escape analysis for detecting upwards-escaping objects. This algorithm is shown in detail in figure 2.7.

[2]An array does not need this size restriction. An uninitialized pointer used as an array reference will be caught by the array bounds checker since the array will have no known size expression.

```
DSG(F)    : Bottom-up data structure graph for function F
ReachableNodes(N, F): DS Nodes reachable from Node N in DSG(F)

for (each function F in program M)
   for (each DSNode N in DSG(F))
       if (N is pointed to by an argument or return value of F or global )
           for (each DS node N' in ReachableNodes(N, F))
              if (N' contains an 'S' (stack) flag)
                  Report ''Rule P3 violated by N'''
```

Figure 2.7: Stack Safety Algorithm

2.7.3 Heap Safety

The most challenging aspect of guaranteeing memory safety without runtime checks or garbage collection, is guaranteeing safe dereferencing of pointers to the heap in the presence of explicit deallocation. Our approach relies on the type homogeneity principle, developed in prior work as discussed in section 2.3.3. We reiterate the principle here:
(Type homogeneity principle) If a freed memory block holding a single object were to be reallocated to another object of the same type and alignment, then dereferencing dangling pointers to the previous freed object cannot cause a type violation.

As discussed in section 2.3.3, this principle can be exploited by applying it to ensure program safety after the pool allocation transformation. The key idea is that we do not free memory from a pool to the system when the memory is deallocated. This enforces that memory in a pool is reused for other allocations in the same pool, but not across pools. This was achieved by simply modifying the runtime memory to only return memory to the system upon `pooldestroy`, but reuse memory released upon `poolfree` for other objects allocated in the pool. With this change, any reference via a dangling pointer to a pool object will be guaranteed to reference either the original object or a new object of the same type and alignment as the original, and *belonging to the same pool*. This ensures that the basic principle described above is satisfied. This approach however, can result in an increase in memory consumption, which we examine in detail in section 2.7.3. As noted in section 2.3.3, applying the type homogeneity principle after automatic pool allocation preserves the pointer analysis results.

Our technique, which exploits the type homogeneity principle, can result in dangling pointers dereferencing stale data or reallocated memory. However, it is important to note that the dangling pointer can only access memory within the same thread. Since a pool only contains memory from a single thread, the memory can only be reallocated to the same thread. Therefore, in the event of a dangling pointer dereference in the core component, the dangling pointer can only access memory within its own thread

and cannot inadvertently access data in a non-core component. This is vital to the property verified in the following chapter (chapter 3): explicit communication between core and non-core components (threads).

Detecting Potential Increases in Memory Consumption

The key issue with our approach to heap safety is memory consumption. The change to the pool allocation runtime library above prevents reuse of memory between two simultaneously live pools. This can have the same disadvantage as the naïve type-based pools — the memory requirement of the program could increase. Note, however, that our pools are much more short-lived than in the naïve approach and are tied to dynamic data structure instances in the program, not static types. We expect, therefore, that during the lifetime of a pool, the most important reuse of memory (if any) is *within* the pool rather than between the pool and other pools. Only the latter causes any potential increase in memory consumption. Nevertheless, any such increases are likely to be of significant concern to programmers of embedded systems.

The goal of our further analysis is to distinguish the situations outlined above, and inform the programmer about data allocation points where potential memory increases can occur. We can classify each pool P into one of three categories:

Case 1 (No reuse): Between any `poolfree` for pool P and the `pooldestroy` for P, there are no calls to `poolalloc` from any pool including P itself. In this case, there is no reuse of P's memory until P is destroyed. Figure 2.8(a) illustrates this situation. Note that all `poolfree` calls to P can be *eliminated as a performance optimization*. This is essentially static garbage collection for the pool since its memory is reclaimed by the `pooldestroy` introduced by the compiler.

Case 2 (Self-reuse): Between any `poolfree` operation on pool P and the call to `pooldestroy` for P, the only `poolalloc` operations are to the same pool P. In this case, the only reuse of memory is within pool P, and the explicit deallocation via `poolfree` ensures that no increase in the program's memory consumption will occur. This is illustrated in Figure 2.8(b): after the first `poolfree` on `p1` there are new allocations in pool `p1` (via the function `addItems`), but not by any other pool.

Case 3 (Cross-reuse): Between the first `poolfree` operation on P and the `pooldestroy` for pool P there are `poolalloc` operations for other pools. Pool `p1` in Figure 2.8(c) falls in this category because there are allocations from pool `p2` via the call to `addItems(p2,t)`. Our transformation in this case may lead to increased memory consumption, and we require this to be approved by the programmer via a compiler option. In such situations,

```
(a) No reuse (case 1)
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeSomeItems(p1,t);


}
freeTree(p1,t);
poolDestroy(p1);

(b) Self-reuse (case 2)
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeSomeItems(p1,t);
   addItems(p1,t);// self-reuse

}
freeTree(p1,t);
poolDestroy(p1);

(c) Self and cross-reuse (case 3)
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeSomeItems(p1,t);
   addItems(p1,t);  // self-reuse
   addItems(p2,t);  // cross-reuse
}
freeTree(p1,t);
poolDestroy(p1);
```

Figure 2.8: Example illustrating 3 types of reuse behavior for a pool `p1`.

the programmer would first analyze or profile the memory consumption of the code, focusing on data structures assigned to Case 3 pools identified by our classification algorithm. Typically the programmer has the following choices:

(a) Often, the increase in memory with Case 3 pools is acceptable. These are cases where there is limited wasted memory from pools with overlapping lifetimes, in spite of not freeing memory back to the system (possibly due to a lot of pool memory self-reuse).

(b) In some situations, the source code of the program could be restructured to avoid Case 3 pools. For instance, since our calls to `poolinit` and `pooldestroy` are at the entries and exits of functions, enclosing the use of a data structure from the point it is first used till its last use within a function potentially moves the `pooldestroy` for the pool

earlier in the program. However, Case 3 pools are sometimes unavoidable if there are long-lived data structures with overlapping lifetimes.

Furthermore, standard software engineering practices tend to minimize the number of Case 3 pools. Examples include separating long-lived and short-lived data into distinct data structure instances, avoiding long-lived pointers to short-lived data, and modular program design (especially confining data structure instances within functions). These observations are supported by our experimental results, which show that Case 3 pools occur in few of our benchmarks, and the increase in memory consumption is small.

Note that the pool in our running example of Figure 2.6 has only self-reuse, and we can guarantee memory safety without any increase in memory consumption. Our experiments in Section 2.9 have produced very few instances of Case 3; they occurred only in 3 out of the 20 embedded and control programs we examined, and none had significant increased in memory consumption due to the change to the pool runtime library.

Amongst the three cases, Case 2 pools are the only pools where there is self-reuse, which result in a dangling pointer potentially dereferencing reallocated memory. In conservatively designed embedded systems, only case 1 and case 3 pools are permitted, completely eliminating the potential dangling pointer dereferences.

Compiler Algorithm for Categorizing Pools

We have developed a compiler analysis to categorize pools into the three cases described above. This algorithm is run after the Automatic Pool Allocation transformation as shown in Figure 2.9, and identifies to which group each pool belongs. For this static analysis, each call to `poolinit()` is a distinct pool. When analyzing a particular function, each distinct pool descriptor (which may be a formal argument or a call to `poolinit()`) is treated as a potentially distinct pool.

Categorizing pools requires analyzing the potential order of execution of pool operations *across the entire program*, using an interprocedural control flow analysis. Automatic Pool Allocation records information about the pools used in each function and the locations of calls to `poolalloc`, `poolfree` and `pooldestroy` inserted for each pool. Pool pointers are passed between procedures but they are not otherwise copied and their address is never taken, so each pool pointer variable declared within a function identifies a distinct pool.

The algorithm for identifying and categorizing reuse within and across pools is shown below:

```
FreeSites(F,P) : set of call sites in F that may call poolfree on pool P
```

```
                directly or indirectly
AllocSites(F,P): set of call sites in F that may call poolalloc on pool
                P directly or indirectly
PoolsFreed(F)  : set of pool arguments of F that may have a poolfree in
                F or one of its callees
PoolsAlloced(F): set of pool arguments of F that may have a poolalloc in
                F or one of its callees
// Analyze direct and indirect calls to poolalloc, poolfree and
// pooldestroy and classifies pools in a function
AnalyzeFunction(Function F)
begin
    BBfreesBefore(BB) : set of pools freed but not destroyed on some
                        path to the beginning of basic block BB
    BBfreesAfter(BB)  : set of pools freed but not destroyed on some
                        path to the end of basic block BB
    BBdestroys(BB)    : set of pools destroyed in basic block BB
    for (each basic block BB in F)
        initialize BBfreesAfter(BB) with pools freed in BB
        initialize BBdestroys(BB) with pools destroyed in BB
        BBfreesAfter(BB) = BBfreesAfter(BB) - BBdestroys(BB)
    while (change) // forward propagate frees on pools, kill free upon
                   // destroy
        for (each basic block BB in F in a reverse post-order traversal
             of the CFG)
            BBfreesBefore(BB) = Union of BBfreesAfter(pred(BB)) for
                                all predecessors of BB
            BBfreesAfter(BB) = BBfreesAfter(BB) UNION
                               (BBfreesBefore(BB) - BBdestroys(BB))
            Recompute change
    // Classify pools as Case 1, 2 or 3
    for (each call site AI in AllocSites(F,P))
        AIBB: basic block corresponding to AI
        BBdestroysBeforeAI = set of pools destroyed before AI in AIBB
        BBfreesBeforeAI =  BBfreesBefore(AIBB) UNION
                           (set of pools freed preceding AI in AIBB)
        for (Pool P1 in (BBfreesBeforeAI - BBdestroysBeforeAI))
            if (P1 == P) Add (F, P1) to ‘‘Case 2 Pools’’
            else         Add (F, P1) to ‘‘Case 3 Pools’’
    if (!(Case 2 or Case 3))
        Add (F, P) to ‘‘Case 1 Pools’’
end;
// Propagate calls to poolalloc and poolfree interprocedurally and
// analyze pools in each function
AnalyzeProgram(Program M)
begin
    for (each SCC in CallGraph of M in post-order)
        while (change = true)
            change = false
            for (each function F in the SCC)
                // Compute AllocSites(F,P), FreeSites(F,P),
                // PoolsFreed(F) and PoolsAlloced(F)
                for (each pool pointer variable P in F)
                // formal argument or local variable
                    for (each call site CS in F that has P as an argument)
```

```
                        for (each function CalledF that can be called
                             at CS)
                            if (CalledF is poolfree for P OR
                                PoolsFreed(CalledF) contains P)
                               if (FreeSites(F,P) does not contain CS)
                                   change = true
                                   add CS to FreeSites(F,P)
                                   if (P is an argument of F)
                                       add P to PoolsFreed(F)
                            if (CalledF is poolalloc on P OR
                                PoolsAlloced(CalledF) contains P)
                                if (AllocSites(F, P) does not contain CS)
                                    change = true
                                    add CS to AllocSites(F,P)
                                    if (P is an argument of F)
                                        add P to PoolsAlloced(F)
        for (each function F in the SCC)
            AnalyzeFunction(F)
end;
```

We say a function F (or a call site C) indirectly calls a pool operation (e.g., `poolfree`) if it calls some function that may directly or indirectly call that operation. The sets `FreeSites(F,P)` and `AllocSites(F,P)` respectively identify the call sites within function F that directly or indirectly invoke `poolfree` and `poolalloc` on pool `P`. The sets `PoolsFreed(F)` and `PoolsAlloced(F)` respectively are sets of incoming pools (i.e., formal pool pointer arguments to function F) for which F may directly or indirectly call `poolfree` or `poolalloc`.

Consider first a single-procedure program containing calls to `poolfree`, `poolalloc` and `pooldestroy`. The analysis then traverses paths from a `poolfree` on a pool to the `pooldestroy` calls on that pool, looking for all calls to `poolalloc` that appear on such a path. This is shown as routine `AnalyzeFunction` in the algorithm. `AnalyzeFunction` contains an iterative forward dataflow algorithm, which, for each program point, computes the set of all pools that are freed but not destroyed along some path to the point. For each basic block `BB`, the sets `BBfreesBefore(BB)` and `BBfreesAfter(BB)` represent these sets at the entry and exit of the block. The set `BBfreesBefore(BB)` is simply a union of the sets `BBfreesAfter(p)` for all predecessor blocks `p` of `BB`. A pool in the set `BBfreesBefore(BB)` is propagated to `BBfreesAfter(BB)` unless there is a call to `pooldestroy` on that pool in `BB`. Each iteration is a linear-time traversal of the basic blocks, and we have found that there are only a small constant number of iterations in practice (as expected because this dataflow problem has the acyclic propagation property [1]). Every `poolalloc` call is then analyzed by checking for calls to `poolfree` on undestroyed pools on any path preceding it. This is computed using `BBfreesBefore` for the basic block corresponding to the `poolalloc` call and the set of pools freed but

not destroyed in the basic block before the `poolalloc` call. Pools are categorized based on the instances of `poolfree` (if any) found on such paths.

Consider next an input program without recursion. The algorithm then makes a bottom-up traversal of the call graph, computing the four kinds of sets above for each function. The bottom-up traversal ensures that the sets `PoolsFreed(C)` and `PoolsAlloced(C)` will be computed for all possible callees C of a function `F`, before visiting `F`. To compute the sets for F, we visit each call site S in F and add this call to `FreeSites(F,P)` if it causes an invocation of `poolfree(P)`, and to `AllocSites(F,P)` similarly. We also add each pool so encountered to `PoolsFreed(F)` or `PoolsAlloced(F)`. We assume that `pooldestroy` on a pool is only called at the function in which the pool is created and hence we do not need to propagate these calls interprocedurally. We can now invoke `AnalyzeFunction(F)` directly to classify all pools in `F`. Note that `AnalyzeFunction(F)` makes no distinction between local and indirect calls to `poolfree`/`poolalloc` for pool `P` since both kinds of call sites are included in `FreeSites(F,P)` and `AllocSites(F,P)`.

To handle recursive and non-recursive programs uniformly, we actually perform the bottom-up traversal on the Strongly Connected Components (SCC's) of the call graph. Within each SCC, we use a simple iterative algorithm in which the sets are propagated from a function to its call sites *within* the SCC until the sets `FreeSites(F,P)` and `AllocSites(F,P)` stabilize for all functions `F` in the SCC and every pool `P`. Once they have stabilized, the sets can be propagated from each function in the SCC to every call site of that function outside the SCC. `AnalyzeFunction` is then applied to each function `F` in the current SCC as explained earlier.

2.8 Array Safety

Array bounds violations are a very cause for memory safety violation. The language restrictions and accompanying static analyses to detect array bounds violation are a part of Dhurjati's dissertation [30] and are summarized here for completeness. Dhurjati has evolved the following restrictions for array operations in order that they can be checked statically:

On all control flow paths,

(**A1**) The index expression used in an array access must evaluate to a value within the bounds of the array.

(**A2**) For all dynamically allocated arrays, the size of the array must be a positive expression.

(**A3**) For every reference of an array `A`, either the index expression in the array reference must be a provably affine transformation of the size

of A or the following must hold.

(a) the array reference has to be inside a loop;

(b) the index expression in the array reference must be a provably affine transformation of the vector of index variables of enclosing loops;

(c) the bounds of the enclosing loops must be provably affine transformations of the size of A and outer loop index variables or vice versa; and

(d) if the index expression in the array reference depends on a symbolic variable s which is independent of a loop index variable (i.e., appears in the constant term in the affine representation), then the memory locations accessed by that reference have to be provably *independent* of the value of s.

(**A4**) A set of trusted library routines with specified preconditions may be used, and arguments passed to those routines must satisfy the preconditions.

(**A5**) The last element of a character array cannot be accessed by the program (trusted library calls like strlen can access it).

The main reason for rules A1 to A3, which enforce that all array accesses are affine, is that static bounds analysis is fundamentally limited by the tractability of analysis of constraints on symbolic integer expressions. A4 enables us to allow trusted string and I/O library routines that make use of arrays. A5 ensures that string routines will not read beyond the size of the array by initializing the last character in any array of characters to be null.

The array bounds analysis accompanying these rules is an interprocedural context-sensitive constraint propagation algorithm that generates a set of constraints on which each array index expression is dependent, until it discovers the size of the array. Then, these set of constraints are inputted to an integer constraint solver like Omega [57]. The constraint solver reports if the index expression is safe, i.e., within the bounds of the array.

2.9 Results

We now evaluate the effectiveness of our above techniques for guaranteeing memory safety on a broad class of embedded and control system benchmarks. We address the following questions:

1. How much effort is required to convert existing embedded programs to conform to our semantic restrictions?

2. Are the pool allocation transformation and heap safety analysis powerful enough to enforce pointer and heap safety statically in different embedded programs?
3. How often do we encounter pools from each of the three categories in these programs?
4. How much does the heap safety transformation affect the execution time and the memory usages of the programs ?
5. Are the semantic restrictions and static analyses for stack safety sufficient for existing embedded programs?
6. Are the array restrictions and bounds-checking algorithm flexible enough to permit existing embedded programs (without extensive changes)?

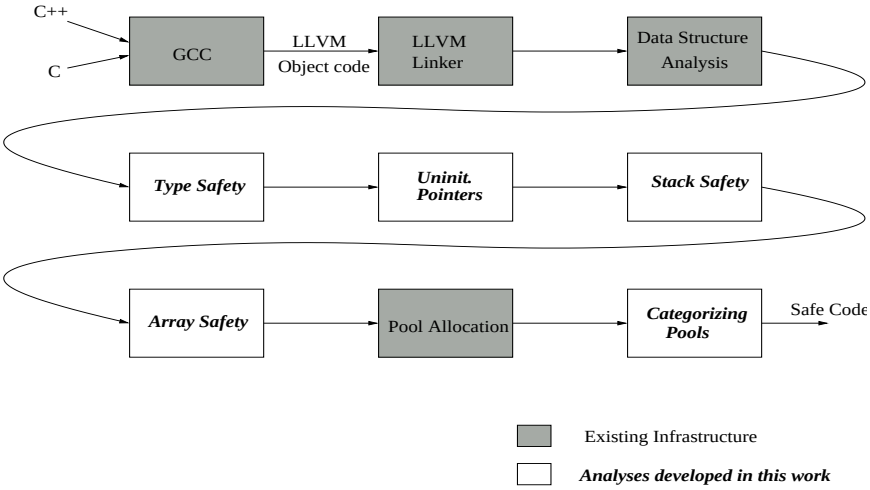


Figure 2.9: Implementation of Analyses

2.9.1 Implementation

We have implemented a safety checking compiler that includes all the compiler techniques described in this paper, using the LLVM compiler infrastructure. Figure 2.9 is a high level block diagram showing the sequence of steps we use to enforce safety. Previously existing compiler components are shown by shaded boxes and the rest are new components developed for the complete system developed with D.Dhurjati. We have also modified our runtime pool allocation library so it does not release free memory in a pool back to the system heap until the pool is destroyed.

2.9.2 Methodology and Porting Effort

Our test programs were derived from two embedded application benchmark suites: 13 from MiBench [48] and 4 from MediaBench [65] (Of the other programs in MediaBench, two fail pool allocation and one is not accepted by

Benchmark	Lines of Code	Lines of Code Modified for type safety	Lines of Code Modified for array safety
control			
Pendulum	300(Average)	0	0
Pendubot	1300(Average)	0	31
TinyOS apps	300(Average)	0	0
automotive			
basicmath	579	1	3
bitcount	17	5	0
qsort	156	0	1
susan	2122	1	0
office			
stringsearch	3215	0	3
security			
sha	269	0	1
blowfish	1502	1	5
rijndael	1773	3	6
network			
dijkstra	348	0	0
telecomm			
CRC 32	282	0	1
adpcm codes	741	0	0
FFT	469	0	0
gsm	6038	0	0
multimedia			
g721	1622	11	0
mpeg(decode)	9839	0	0
epic	3524	7	0
rasta	7373	25	0
Totals: 20	41769	70	53

Table 2.1: Benchmarks, code sizes, and source changes

the current LLVM C front-end), two classes of experimental control codes, and sensor network applications. MiBench consists of embedded programs from a variety of domains including telecommunications, security, networking, etc. MediaBench are predominantly multimedia programs. In addition, we tested a set of PID controllers for an inverted pendulum running on the Simplex real-time architecture [81], LQR state space controllers for the Pendubot experiment from the controls laboratory at the University of Illinois, and real-time sensor applications in sensor networks running on TinyOS [52]. We believe that these programs cover a wide variety of embedded applications used in practice. The applications that our LLVM C front-end or the pool allocation pass currently refuse to compile are similar to the ones that we report here with respect to code size (except ghostscript) and in usage of dynamic memory and we do not expect the results to change qualitatively for the remaining benchmarks.

The program `rasta` used a library called `libsphere` whose source was

Benchmark	Heap and Pointer safety (Case)	Stack safety	Array safety
control			
Pendulum	Yes	Yes	Yes
Pendubot	Yes (Case 1)	Yes	Yes
TinyOS apps	Yes	Yes	Yes
automotive			
basicmath	Yes	Yes	Yes
bitcount	Yes	Yes	Yes
qsort	Yes	Yes	Yes
susan	Yes (Case 1)	Yes	No
office			
stringsearch	Yes	Yes	Yes
security			
sha	Yes	Yes	Yes
blowfish	Yes	Yes	Yes
rijndael	Yes	No	Yes
network			
dijkstra	Yes (Case 2)	Yes	No
telecomm			
CRC 32	Yes	Yes	Yes
adpcm codes	Yes	Yes	No
FFT	Yes (Case 1)	Yes	No
gsm	Yes (Case 1,2)	Yes	No
multimedia			
g721	Yes	Yes	No
mpeg(decode)	Yes (Case 1,3)	Yes	No
epic	Yes (Cases 1,3)	Yes	No
rasta	Yes (Cases 1,3)	Yes	No
Totals: 20	20	19	11

Table 2.2: Benchmarks and analysis results

not available. The experiments for `rasta` assumed that this library is safe and checked the safety of the available source. Also, for each of the programs above, we designate library and system calls (file reading and writing routines, for instance), whose source is unavailable, as being trusted. This is safe since we manually enforce that each of these programs is linked only to trusted libraries.

The benchmarks, their sizes, and our results for each are shown in Table 2.1 and 2.2.

We found that a few lines of code had to be changed in several benchmarks to conform to our rules, particularly for type safety and array safety. These are shown in the third and fourth columns of Table 2.1. The largest changes were for rule (**T3**) in `rasta`, `Pendubot`, and `g721`. All the three programs used unions with incompatible types; `rasta` had a union with a float and an array of four chars to swap the bytes of the float value, and `g721` did the same for an unsigned int. We rewrote the code using shift

Benchmark	Execution Time (secs)		
	Orig time	heap safety time	exec ratio
automotive			
basicmath	1.667	1.672	1.00
bitcount	0.710	0.727	1.02
qsort	0.405	0.404	1.00
susan	0.670	0.675	1.01
office			
stringsearch	0.024	0.024	1.00
security			
sha	0.145	0.138	0.95
blowfish	0.713	0.722	1.01
rijndael	0.340	0.366	1.07
network			
dijkstra	0.340	0.349	1.02
telecomm			
CRC 32	1.463	1.53	1.04
adpcm codes	1.255	1.252	1.00
FFT	0.495	0.478	0.96
gsm	1.979	1.959	0.98
multimedia			
g721	0.354	0.355	1.00
mpeg(decode)	0.331	0.320	0.97
epic	0.126	0.128	1.01
rasta	0.124	0.125	1.01

Table 2.3: Execution time and memory usage for heap safety approach. exec ratio is the ratio of execution time after pool allocation to the original time. (A ratio of 2 means the program runs twice as long as the original)

operations and eliminated the union. The benchmark `epic` used a wrapper around malloc to check the return value of malloc and to exit the program if it were null. This resulted in casts from `char*` to a pointer to the type being allocated after each call to this wrapper function. We replaced the wrapper function with a plain malloc call in order to prevent these casts. The return value check however is preserved since these calls to malloc are converted into `poolalloc` and the return value of malloc is checked by the `poolalloc` library function. The other changes for type safety were very small. For instance, we needed to initialize local pointer variables before use within their parent function. Also, the `fread` and `fwrite` system calls take a `char*` value as their first argument, leading to a cast from an arbitrary pointer to a `char*` before it is passed as argument. This violates rule (**T2**) and also prevents pool allocation from being applied to the object being passed in. We have defined separate trusted wrapper functions around `fread` and `fwrite` for each primitive non-pointer type and we changed the source code to use the appropriate wrapper functions. (Programs that read non-primitive data from a file would be rejected.)

Benchmark	Memory Usage (bytes)				
	Orig mem usage	pool alloc mem usage	mem ratio 1	pool alloc + safety restriction	mem ratio 2
automotive					
basicmath	16384	16384	1	16384	1
bitcount	16384	16384	1	16384	1
qsort	24576	24576	1	24576	1
susan	253952	253952	1	253952	1
office					
stringsearch	16384	16384	1	16384	1
security					
sha	24576	24756	1	24576	1
blowfish	24576	24756	1	24576	1
rijndael	24576	24576	1	24576	1
network					
dijkstra	32768	32768	1	32768	1
telecomm					
CRC 32	16384	16384	1	16384	1
adpcm codes	0	0	-	0	-
FFT	540672	540672	1	540672	1
gsm	24576	24576	1	24576	1
multimedia					
g721	24576	24576	1	24576	1
mpeg(decode)	385024	401408	1.04	401408	1
epic	671744	681616	1.01	779920	1.14
rasta	147456	212992	1.44	212992	1

Table 2.4: Memory usage of the heap safety technique. mem ratio 1 is the ratio of the memory usage of program after pool allocation to that of the original program. mem ratio 2 is the ratio of the memory usage of pool allocated program with our safety restriction to that of just the pool allocated program

For the array safety rules, we had to rewrite a few lines of code in 8 programs. The changes were generally minimal and obvious. For instance, in `blowfish` a command line argument was accessed by iterating and checking if the last character was null, which had to be rewritten to use `strlen()` for the loop bound and using an induction variable in the while loop which depends on the `strlen()`. In another case (search_string), an array of strings was accessed in a while loop with the index variable unrelated to the bounds. We had to rewrite the code to make the access obey our language rules described in Section 2.8.

Besides requiring very few modifications, the changes themselves were simple and local and in most cases obvious from reading the code or from compiler error messages. Overall, we believe the porting effort to use our compiler for standard C programs is small to negligible.

2.9.3 Effectiveness of Pointer and Heap Safety Analysis

The *Heap and Pointer Safety* column in Table 2.2 shows that our compiler was able to enforce safety of heap and pointer usage for *all 20 programs* we studied. More precisely, the DSA and pool allocation techniques together are sufficiently precise to partition the programs heap data into type homogeneous pools (after the few changes to these programs we made to ensure that the programs pass our type safety requirements). About half the benchmarks use no dynamic memory allocation (though they still use pointers). For the other benchmarks, the same column shows the different categories of pools found in each one. The results show that we were able to prove heap safety without increase in memory consumption (i.e., Case 1 or Case 2 pools — no reuse or only self-reuse), for all 13 MiBench benchmarks, 1 of the 4 MediaBench programs, and all the control programs.

Only three programs, `mpeg2decode`, `rasta` and `epic`, have pools with cross-reuse by other pools (Case 3). In practice, our experimental results (see table 2.3, table 2.4 and Section 2.9.3) have shown that these do not result in a significant increase in memory consumption. The three programs, `mpeg2decode`, `rasta` and `epic` make extensive use of dynamic memory, yet they contain very few pools that fall under Case 3: just 4 of the 8 pools in `mpeg2decode`, 3 of 12 in `epic`, and 19 of 80 in `rasta`.[3] In fact, all the case 3 pools in `mpeg2decode`, `rasta`, and `epic` also have self-reuse from the same pool, so that the effect of not freeing memory to other pools is mitigated. We have also observed that some case 3 pools in these three benchmarks can be converted to case 1 or 2 with more sophisticated compiler analyses where the `pooldestroy` on a pool is moved as close to the last `poolfree` on the pool as possible without compromising safety [64].

Another interesting use of dynamic memory is seen in `dijkstra`, where a linked list is live throughout the program and the program repeatedly allocates and deallocates memory. In a language with explicit regions such as Cyclone [47] or RT-Java, this list would have to go on a garbage collected heap or incur a potentially large memory increase. In our technique, the case 2 pool allows reuse of memory within the pool. Finally, there were a number of Case 1 pools, which are amenable to the optimization of turning off individual object frees entirely, effectively performing static garbage collection with no increase in memory usage.

[3]The specific number of pools and numbers of case 1, 2 and 3 pools depend on the precision of DSA and pool allocation. We have made several improvements in DSA and pool allocation since our initial experiments [32], leading to larger numbers of total pools and larger numbers of case 2 and 3 pools in `epic`, `rasta`, and `mpeg2decode`. Nevertheless, the overall results are qualitatively similar, and our new measurements of memory consumption show that the impact on peak memory consumption of these codes is negligible.

Evaluation of Run-time Costs of Heap Safety

In Table 2.3 and 2.4, we present our results on evaluating the run-time costs of the heap safety approach. Since we do not insert any run-time array bounds checks in this work, the programs rejected by our array bounds checker are actually unsafe. We include their execution times only to show the effect of the pool allocation transformation on performance. First, we compared the execution times of these applications after the pool allocation transformation used for heap safety to the original execution time. Most of the execution times after pool allocation are within 2% of the original execution time and only one program shows an increase of 7%. In some cases, we can even see that the pool allocation transformation improves the execution time. These results show that our heap safety mechanism only results in a marginal increase (if any) in execution time.

Next we measured the maximum memory usage of these programs. To identify the causes for increase or decrease in memory consumption, we measured the usage in three versions: original program, pool allocated program, and pool allocated program along with our safety restriction that memory cannot be released to the system until pooldestroy. Our other analyses (stack safety, array safety, etc.) do not change the memory consumption of the program. To better understand the numbers, we first give a brief description of our pool allocation run-time library. The library internally manages memory using malloc and free. To amortize the allocation costs over various allocation requests it mallocs memory in multiples of pages of size 1K bytes. It releases memory to the system if it has more than a threshold number of free pages.

The column "Mem ratio 1" in Table 2.3 shows the memory increase due to pool allocation when compared to the original program. The increase is insignificant in most programs except rasta, where it is 44%. We found that rasta has many global pools which allocated a total memory of 8 bytes while we were reserving a page of 1024 bytes for each such pool in our run-time library. In general, the total memory usage for these embedded programs is not high and any wastage (such as in `rasta`) appears large in percentage terms but its impact is minor in practice. If we decrease the page size to 512 bytes, we found that memory increase reduced to 28%, showing that a well-tuned pool allocation run-time library that dynamically increases page sizes depending on the allocation requests can do much better than our simple untuned version.

We then measured the additional increase in memory usage due to our safety restriction. As mem ratio 2 illustrates, only `epic`, which has a Case 3 pool, shows an increase (of 14%) due to the additional safety restriction. Other programs with Case 3 pools, `rasta` and `mpeg2decode`, do not show any increase.

Overall, our results indicate that Case 3 pools occur infrequently even in complex embedded programs and typically never occur at all in simpler programs. When it does occur, the increase in maximum memory usage seems acceptable. This is strong empirical evidence that our technique is powerful enough to enforce heap safety statically in a broad range of embedded programs.

2.9.4 Effectiveness of Stack Safety Checks

Our stack safety check ensures that pointers to the stack frame in a function are not accessible after that function returns. The stack safety column of Table 2.2 shows that only 1 program (`rijndael`) failed this check. This proved to be a false positive that occurred because Data Structure Analysis is flow-insensitive. In `rijndael`, a pointer to a local variable is stored in a global but the global is reinitialized by a callee of the function before the function returns. Such cases must be handled by restructuring the program.

2.9.5 Effectiveness of Array Access Checks

The array bounds checker developed by Dhurjati [32, 33, 30] passed all the 3 classes of control programs, 8 of the 13 benchmarks from MiBench, and none from MediaBench, after the few changes described earlier. Interestingly, the tests detected 4 potential array bound violations in the MiBench suite and 2 in MediaBench: one each in `dijkstra` (both the large and small versions), `epic`, and `blowfish` and two violations in `g721`. Overall, safety checking of complex array references remains the most significant obstacle to our goal of enforcing memory safety with no run-time software checks for a broad class of embedded applications. As discussed in the following section, extensions to this work have accepted programs that do not adhere to the Control-C language restrictions through run-time checks.

2.10 Extensions to the above work

The restrictions of Control-C make it expressive enough to program a large class of embedded and control systems. However, these restrictions are too onerous for a large class of non-type-safe low-level codes such as operating system codes and other complex application software that use the following constructs:

1. Arbitrary casts: Casting memory between incompatible types, e.g. integer to a pointer

2. Complex array accesses: Non-affine arrays or arrays that are stored on the heap (thus causing them to be non-affine by our rules).

3. Dangling stack pointer: The presence of an argument, return value or global variable that potentially points to a stack location after the function returns.

Significantly, extending the techniques described in this work to non-type-safe codes presents an important challenge. Enforcing the semantics of the pointer analysis and call graph, which is used by the interprocedural safety analyses described in this work, is non-trivial in the presence of non-type-safe code constructs. Thus, any technique attempting to use extensive static analysis to prove memory safety for non-type-safe codes should guarantee sound program semantics. Dhurjati [30] has designed the run-time checks necessary in order to provide memory safety as well as sound semantics for non-type-safe programs. The sound semantics provided can be used by further static analyses on these non-type-safe programs.

2.11 Related Work

There have been several attempts at providing memory safety. These can be categorized into safe languages through compiler analyses, language-level solutions, and purely run-time approaches.

2.11.1 Safe Languages

Embedded systems are mostly written in C and C++. There are several reasons why existing safe languages are unattractive for embedded systems. Virtually all safe languages today e.g. Java [45], Modula-3 [17], Safe-C [6] and CCured [73] rely on garbage collection to ensure that freed memory locations are not dereferenced, along with *run-time software checks* before memory operations such as bounds checks for arrays and null pointer checks. Real-time garbage collection (GC) implementations have been shown to incur an execution and memory overhead [9] (of up to 2.5x to achieve acceptable real-time performance) while non-real-time GC algorithms introduce unpredictable run-time delays and increases in average time and space. The real-time specification of Java [12] avoids GC entire for subsets of the heap by providing three additional types of *MemoryAreas* that are not garbage collected. Of these, *ScopedMemory* type defines nested regions for dynamic allocation. This is more restrictive than our approach and requires run-time checks to ensure that there are no references from objects in an outer scoped region to an object in an inner one. Additionally RT-Java inherits run-time checks for null pointers, array bounds, and type coercions.

All the safe languages above as well as systems like Jones and Kelly [56], which rely on run-time software checks to make C programs safe have prohibitive overheads (upto 500%). Amongst these, CCured has shown the best

run-time results so far. CCured extends the C type system to statically infer safe pointers and adds run-time checks for the other pointers. CCured still uses garbage collection for heap safety and uses pointer meta-data to perform run-time checks which requires porting effort for library compatibility. SafeC uses a fat pointer representation o store spatial and temporal information for all pointers and uses run-time software checks to detect memory errors. They report execution overheads of up to 540% and space overheads of up to 100%.

2.11.2 Language-level Approaches

Region-based languages achieve safety without GC [89, 40, 13, 28], but have two problems:(a) they require significant porting effort due to the use of program annotations, and (b) they provide no mechanisms to free or reuse memory within a region, so data structures that shrink and grow with non-nested object liefe times must be put in a separate garbage collected heap or incur a potentially large increase in memory consumption. Both Cyclone and RT Java both include a separate garbage collected heap. Boyapati *et al* [13] present a static type system combining ownership types with region types, to eliminate the run-time checks needed for ensuring safe region deallocation in RT Java. As a region-based language, they have the same differences from our work as discussed above. They provide an additional mechanism based on "sub-regions" of a region for sharing region data safely across threads, using reference counts to reclaim the data. We do not support multi-threaded applications so far.

Linear types and alias types [25, 92, 28, 38] have been used to prove memory safety statically in the presence of explicit deallocation of objects. They achieve this primarily with severe restrictions on aliases in a program, which so far have not proved practical for realistic programs. One of these languages, Vault [28], also uses such a type system (much more successfully) to encode many important correctness requirements for other dynamic resources within an application (e.g., file handles and sockets). It would be very attractive to use Vault's mechanisms within our programming environment to check statically key correctness requirements of system calls and trusted libraries.

2.11.3 Run-time Techniques

The most significant run-time approach to memory safety has been software fault isolation (SFI) [91], which ignores all language-level information and enforces memory safety by *sand-boxing* every memory access/jump at run time. SFI does not detect semantic errors such as array bounds errors, references to uninitialized values, or accesses to locations in dead stack

frames. Furthermore, SFI introduces significant and sometimes high run-time overhead, ranging from 25% to 59% when checking only write references and typically over 100% when checking both reads and writes.

A valuable orthogonal strategy for compiler-based secure and reliable systems is Proof Carrying Code (PCC) [72]. The benefit of PCC is that an unreliable safety checking compiler can be untrusted, and only a simple proof checker (which can be made much more reliable) is required within the trusted code base. Fundamentally, PCC does not change which aspects of a program require static analysis and which require run-time checking — that still depends on the language design and compiler capabilities. Thus, PCC is orthogonal to our work, and could be valuable for taking our safety-checking compiler outside the trusted code base.

2.12 Chapter Summary

In this chapter, we described Control-C, a semantically restricted subset of C, which is expressive enough to program a broad class of embedded and control systems, but is guaranteed to be memory safe through our static analyses without garbage collection or run-time software checks. Memory safety guarantees fault containment within a software component with respect to memory errors that frequently occur in type-unsafe languages like C and C++. In addition, our definition of memory safety prevents execution of illegal system calls from the data area, thus preventing malicious components from executing "manufactured" code and hidden system calls. This enables us to check that an unreliable component only calls trusted system calls and require that the source of the entire program is available

Our most novel contribution is our heap safety technique where, through the type homogeneity principle, we enforce that dangling pointer dereferences do not cause memory safety violations. This technique prevents safety violations due to dangling pointers while allowing explicit deallocation (i.e. no automatic memory management). While our technique potentially results in an increase in memory consumption, we established the specific patterns, where this can actually occur.

We combine our heap safety analyses with a set of language rules for enforcing basic type safety, restricted array accesses (the affine restrictions), and stack safety. Together, we tested our analyses on 20 programs in the embedded and control systems domain and found that we were able to analyze heap safety for *all these programs*. Only three programs potentially had an increase in memory according to our case classification algorithm and only one program (`epic`) had a small increase in practice. stack safety checking failed on one program. The array rules were violated by 9 programs. Thus, static array bounds analysis remains the main bottleneck for

static memory safety. We currently require run-time checks for all array accesses that cannot be statically proven safe.

To extend our programs to an even broader class of embedded systems written in a type-safe subset of C, two run-time checks may be necessary: uninitialized pointers if no reserved address space is available and array bounds checking for non-affine array indexing. Both of these result in run-time overheads. In the case of uninitialized pointers we are unable to take advantage of hardware checks and require run-time software checks. For non-affine array indexing, the index should be checked at run-time before being used.

2.13 Future Work

Thus far, we have developed compiler techniques to prove memory safety for single-threaded programs which follow Control-C language rules. Extensions to this work have implemented run-time checks for the general class of C programs. These can be found in [30]. Two directions for future work in this realm are enumerated below:

1. Memory safety of multi-threaded programs - We plan to extend our current techniques to multi-threaded programs where we guarantee that memory errors in one thread cannot corrupt the functioning of the other threads in a program.

2. Memory safety of incomplete programs - Currently, we require the whole program source to be available in order to guarantee memory safety. It would be useful to guarantee memory safety of a programs, which do not contain full source code. This will necessitate annotations for the missing functions in the program source.

Chapter 3

SafeFlow: Safe Value Flow in Embedded Control Systems

In chapter 2, we discussed our techniques to ensure that memory errors in an unreliable component are contained within the component or result in safe run-time failures, but do not corrupt other components. However, control systems consist of critical components which communicate with unreliable or untested components and utilize the data generated by these components. Simply enforcing memory safety does not prevent errors due to communication of erroneous values across components. Such communication can be achieved through trusted library calls or code that accesses shared memory. Erroneous values thus propagated from an unreliable component to a critical component can lead to critical functionality failure. While fault containment through partitioning (e.g. memory safety) is a well-understood problem, erroneous value propagation through trusted communication channels typically result in fault propagation across components. In chapter 1, we observed that the separation principle required safe value flow from non-core to core components.

We first illustrate a typical control system and show that complete isolation of core and non-core components is impractical using an inverted pendulum control system (figure 3.1). The pendulum is balanced by the controller, which periodically reads the track position and pendulum angle from the sensor and outputs a voltage value between -5V and $+5$V to the actuator moving the supporting trolley left or right with different accelerations. The critical functionality of the system is to keep the pendulum upright at all times. This can be achieved by the core system consisting of the core controller and the sensor/actuator implemented in hardware. A controller, which minimizes the jitter of the pendulum while balancing the pendulum using more complex control algorithms, is implemented separately as a non-core component, which communicates with the core controller using shared memory. Similarly any user interface, which displays the status of the controllers by reading the shared memory is a non-core component.

In general, there is a two-way data flow between core and non-core components. For instance, in Figure 3.1, the core controller communicates sensor readings to the non-core controller, which, in turn, communicates its

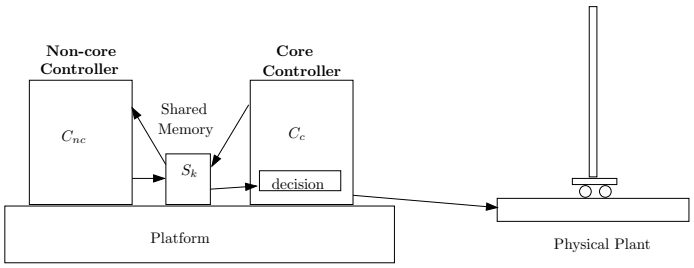


Figure 3.1: Running Example: Inverted Pendulum

computed control outputs to the core controller. A unique attribute of values in control systems is that they quantify properties of the physical world. This hybrid nature of control systems, due to the continuous dynamics of the plant and the environment ensures that values exhibit continuity. This enables the core subsystem to maintain a conservative model of the physical system, which can be used to verify safety and system recoverability properties of unreliable values generated by the non-core subsystem. As in any other domain, it is difficult, if not impossible, to check the correctness of values such as control outputs (e.g. output voltage of the inverted pendulum controller) if they are in a permissible range (e.g. [−5,+5]V). However, the core subsystem can take advantage of the model to verify that the system remains in a recoverable state if a non-core value is utilized by the core subsystem. This kind of a safety check is termed a *monitor*.

Examples of such monitors abound in control system design. In Figure 3.1, the core component can use a stability envelope, represented by a Lyapunov function within the admissible states, proposed by the Simplex architecture [82] as a run-time monitor to check that the system remains in a recoverable state if a non-core control output is applied to the plant. The ability to check recoverability has been exploited by Cunha *et al* [26] for disaster prediction and avoidance in control systems. In our own experience with autonomous car controllers at UIUC [46], control outputs are monitored for potential collisions with other cars or obstacles before being applied to a car actuator.

The broad goal of this chapter is to enforce the following property in the actual system implementation (in C):

> **Safe Value Flow Paths**: *All non-core values flowing into a core component should be monitored before use in critical computation*

This verifies that the core subsystem does not have any unmonitored value dependencies on non-core values (The correctness of the monitor itself is assumed and verifying it is outside the scope of this work). Ding and Sha have described this requirement as a prerequisite to the core component *using* the non-core component, but not *depending* on it [34]. This is a re-

finement of the conventional notion that there exists a failure dependency if there is *any data flow* between two component [69].

Lampson has defined three channels of value flow between components in the security context: legitimate, covert, and storage [61]. While legitimate channels of information involve communication through variables in the local process context, storage channels use environment objects and covert channels use means that are not reflected in values stored anywhere in the system (e.g. execution time). In this work, we only address legitimate channels of value flow from non-core to core components. Particularly, through the work, we assume that all communication is performed through shared memory (Message passing extensions are briefly discussed in section 3.2.4). This is a reasonable assumption since shared memory communication is frequently used in embedded control systems for its fine-grained flexibility and the potential run-time efficiency. While covert channels are not relevant in the context of dependability, value flow through storage channels needs to be addressed in future work.

There are three distinct sources of low-level implementation errors that can violate the safe value flow property above. Firstly, programming bugs can cause the core component to use non-core values without monitoring along some path. This is particularly important as the number of paths in the core component increases, making it difficult to inspect manually. Secondly, programmers can use unmonitored non-core values due to inadvertent accesses to non-core values in shared memory. The primary reason for such accesses is that, in C, pointers to shared memory are not locally distinguishable from other pointers. Interprocedural propagation of shared memory pointers can result in inadvertent accesses to shared memory locations in the core component. The third source of implementation errors is subtler in nature. Core component developers often make assumptions regarding the safety of certain shared memory locations and rely on the encapsulation of the non-core component, synchronization and atomicity, data format compatibility and other pre-conditions that are difficult to verify. Previous work that verifies these properties (see section 3.4) have been best-effort in nature. Violation of any of the above properties results in the propagation of unreliable values to the core component that should be run-time monitored. In this sense, the safe value flow property is foundational and offers a "last line of defense" against many difficult-to-detect interaction errors.

We have designed *SafeFlow*, a static analysis tool that detects potential value dependencies of core components on the non-core subsystem. Static analysis offers the benefits of incurring no run-time overheads and early error detection, which are attractive advantages for embedded systems (run-time error dependency detection incurs performance penalties). Assuming that monitors are correctly implemented, SafeFlow relies on a

few simple and local programmer annotations to describe semantic information about monitors, critical data, and shared memory initialization in the core component. Also, C programs have many language features that make them difficult to analyze statically. Hence, SafeFlow takes advantage of the domain-specific uses of shared memory and imposes a few reasonable semantic restrictions on shared memory pointer usage.

The SafeFlow analysis precisely identifies all uses of unmonitored non-core values communicated through shared memory to core components at development time. Importantly, it reports errors for all unmonitored non-core values that affect critical data within the core component. A secondary contribution of SafeFlow is the simple and succinct annotation language that is useful in representing semantic information in embedded C programs. We applied SafeFlow to three prototype control systems and discovered five critical erroneous value dependencies of the core subsystem on non-core values. There are two main limitations to SafeFlow. First, SafeFlow can produce a few false positives due to control dependence on non-core values not used in critical data computation and due to imprecision of the static analysis. We currently require that the errors are verified using the value flow graphs manually. Secondly, erroneous annotation of an inaccurate or incomplete monitoring function can cause it to miss real dependencies (false negatives). This problem is nearly impossible to eliminate, though we mitigate it by designing a simple annotation language.

3.1 Basic Approach

Consider a core component, which communicates with non-core components using a set of shared variables. For each shared variable, S_i, we define the following mutually exclusive predicates:

- `noncore`(S_i) : holds if x can be written by any non-core component
- `core`(S_i) : holds if it can be verified that x is only written by core components. (`core`(S_i) $\Rightarrow \neg$ `noncore`(S_i) and `noncore`(S_i) $\Rightarrow \neg$ `core`(S_i))

Strictly applying the `noncore` predicate to shared variables that can be written by non-core components helps us detect dependencies on non-core values that arise due to difficult-to-detect bugs in inter-component interaction such as data format compatibility and synchronization. For a local value in the core component, x, we define the following mutually exclusive predicates:

- `safe`(x) : holds if x is defined by the core component or if its value is not dependent on any non-core values
- `unsafe`(x) : holds if x is dependent on any non-core values. (`unsafe`(x) $\Rightarrow \neg$`safe`(x) and `safe`(x) $\Rightarrow \neg$ `unsafe`(x))

We now define the following operational rules for shared variable access in core components:

- Shared memory read: For $x = \texttt{read}(S_i)$,
  $\texttt{noncore}(S_i) \Rightarrow \texttt{unsafe}(x)$ and
  $\texttt{core}(S_i) \Rightarrow \texttt{safe}(x)$
  Reading a non-core shared variable returns an unsafe local value. Also, reading a core shared variable returns a safe local value.

- Shared memory write: $\texttt{write}(S_i, x)$
  Writes to a shared variables, S_i, using a local value in the core component *does not modify* the truth values of $\texttt{core}(S_i)$ and $\texttt{noncore}(S_i)$. This is because non-core shared memory locations are assumed to be accessible to non-core components throughout their lifetime. Verifying the absence of data races and the correctness of synchronized accesses is difficult and cannot be assumed.

Using the above operational semantics, all reads of non-core shared variables by the core component only return unsafe values. However, as discussed earlier, core components in control systems contain run-time monitors for non-core values, which read non-core values and check them for safety before using them. In order to handle this, the programmer identifies functions where the non-core shared variable, S_i, is monitored before use (termed a *monitoring function* for S_i), specifying that $\texttt{core}(S_i)$ holds within this function. Thus, using our operational rules, reading S_i within this monitoring function returns safe local values. The programmer is expected to verify that the monitoring function correctly checks the non-core values for safety (or recoverability) before storing it in local variables that escape the monitoring function or using it in computation of critical data. This facility enables core components in control systems to safely read non-core values.

In the following section, we use the above principles to design a simple annotation language, which encodes semantic information in the core component, and statically analyze the core component to detect unsafe non-core value propagation to critical core component data.

3.2 SafeFlow Analysis

In the previous section, we described our basic approach in identifying monitoring functions and using this information to analyze the core component for unmonitored non-core value accesses that propagate to critical data. In this section, we first describe the annotations that we require from the developer, which provide semantic information about monitoring functions and critical data. In order to use this information in our analysis, we need

to address two key issues that arise in weakly typed languages such as C, which makes sound and precise analysis of programs challenging: (a) memory errors and (b) aliasing and type-unsafe language constructs. To address (a), we leverage our prior work on guaranteeing memory safety and enforcing the semantics of the pointer analysis results in the presence of memory errors. To address (b), we impose a reasonable set of restrictions on shared memory pointer usage, by exploiting the limited ways in which embedded control systems use shared memory. These restrictions enable us to statically analyze the core component to precisely identify unmonitored non-core value access as well as erroneous dependencies with few false positives.

```
  SHMData *noncoreCtrl;
  SHMData *feedback;

  float decision(Feedback* f,
                 float safeControl,
                 SHMData *noncoreCtrl)
  /***SafeFlow Annotation
      assume(core(noncoreCtrl, 0,
                  sizeof(SHMData))) ***/

1:   if (checkSafety(feedback, noncoreCtrl))
2:      return noncoreCtrl->control;
3:   else
4:      return safeControl;
  }

  main()
  {
1:   void *shmStart;
     /* Initialize shared memory */
2:   shmid = shmget(SHMKEY, SHMSize, flags);
3:   shmStart = shmat(shmid, 0, 0);
4:   feedback = (SHMData *) shmStart;
5:   noncoreCtrl = feedback + 1;

6:   Lock(shmLock);
7:   while (1)
8:   {
9:      float *output;
10:     getFeedback(feedback);
11:     computeSafety(feedback,
                      &safeControl);
12:     Unlock(shmLock);
13:     wait(tsecs);  /* Wait for non-core
                component to compute value */
14:     Lock(shmLock);
15:     output = decision(feedback,
                   safeControl, noncoreCtrl);
        /***SafeFlow Annotation
            assert(safe(output)); ***/
16:     sendControl(output);
17:   }
  }
```

Figure 3.2: Example: Core Controller Code

In order to explain our analysis, we use the example in Figure 3.2, which is a simplified version of the core controller in the Simplex architecture implementation for the inverted pendulum [81] from figure 3.1. In each

period, the core controller dispatches a control output to the actuator in order to balance the pendulum. The output of the non-core controller is dispatched, if it can be checked to maintain the system in a recoverable state by the `decision` module. Otherwise a safe control output, computed by the core controller, is applied to the actuator. The routine, `main`, allocates the shared memory using the UNIX system call (lines 1-3) and initializes the global variables, `noncore` and `feedback`, to point to shared memory (lines 4-5). Within the loop, during each period, the core component receives feedback about the position of the pendulum, which is published in shared memory (line 10), computes the safe control output (line 11), waits for the complex controller to publish its computed control output (line 13), checks the non-core component output for recoverability in the function, `decision`, and sends the appropriate control output to the actuator (line 16). The goal of our analysis is to detect unmonitored non-core values in the core component and enforce that the critical value in the core component, `output`, does not depend on any unmonitored non-core values.

3.2.1 Annotations

One of our major goals in this work is to incur minimal or no burden on the programmer in verifying safe value flow in the system. Thus, we ensure that our approach requires minimal programmer annotations. Moreover, we require that annotations are local, succinct, and intuitive to the programmer. Our analysis requires two kinds of semantic annotations from the programmer: (a) Identifying critical data and (b) Characterizing monitoring functions. In general, annotations are undesirable since they preclude using the technique on legacy code without some porting effort and incur a burden on the programmer. However, annotations in our approach are unavoidable since they describe semantic information only known to the developer. On the flip side, imposing annotations on the programmer has the advantage that it enforces a discipline with respect to identifying critical data and the behavior of monitoring functions.

Our annotations are enclosed within C comments which begin with the special string, *SafeFlow Annotation*, as shown in figure 3.2. There are two kinds of annotations: `assume` annotations that provide semantic information about monitoring functions that can be used as facts by the analysis and `assert` annotations that specify the property that must be checked or validated by the analysis.

Monitoring functions need to specify that certain memory locations in shared memory can be assumed to be core in the function and in any function invoked recursively by the monitoring function. For this purpose, the `assume` annotation is declared using the predicate `core`, which is applied to a shared memory pointer, `shmptr` as follows:

`core(shmptr, offset, size)` , which denotes that the shared memory locations accessible using `shmptr` from offset, `offset`, for `size` bytes can be assumed to contain core values. Offset and size values should span an entire array in shared memory, since an array is treated as a single unit by our analysis; otherwise, the annotation becomes ineffective.

In other words, the values read from these locations will be safe according to our operational rules in section 3.1. This annotation is illustrated in the function, `decision`, above the function body in figure 3.2, which specifies that the shared memory pointer, `noncoreCtrl`, can be dereferenced safely between offsets `0` and `sizeof(SHMData)`. This annotation is local and can be specified in terms of local or global shared memory pointers. Also, the monitoring function developer clearly knows the locally shared memory locations that are being monitored and used in that function.

The other semantic information we require for our analysis is the identification of critical data in the core component. Here, we need to specify the requirement that this critical data is safe i.e. it does not depend on non-core values. For this purpose, we employ the `assert` annotation on the `safe` predicate, which takes a primitive typed variable such as `char`, `int`, `float`, or `double`). Its syntax is simply the following:
`safe(x)`, which denotes that the local value `x` is safe.
This is illustrated in the annotation preceding line 16 in `main` in figure 3.2. In general, annotations that identify the critical data in the core component are inserted at program points preceding communication with another component through an I/O operation. Similarly, the arguments to system calls such as the process-id argument to `kill` are asserted to be critical data in the core component. The `assert` annotations required by our analysis are much simpler than the assertion invariants in [68], which attempt to capture the functionality of the system (That work has shown that specifying the latter is not easy for all programmers).

In addition to the above annotations, in our implementation, we need some annotations to describe the shared memory pointers returned by untyped shared memory initializing functions. This is described in section 3.2.2.

3.2.2 Language Restrictions

Generally, precisely checking safe usage of unreliable values in components written in weakly typed languages like C is statically undecidable. While C is extensively used in programming embedded system components, it permits many type-unsafe constructs including pointer arithmetic, arbitrary casts and even memory errors, due to which it is impossible to precisely identify shared memory accesses statically in generic C programs.

The first issue posed by type-unsafe constructs and memory errors is that the local pointers in the core component could access shared memory

in arbitrary and unpredictable ways via bounds violation, dangling pointer dereference or uninitialized pointer dereferences. In order to counter this, we propose to leverage our previous work on memory safety. In [33], we proposed a restricted type-safe subset of C, which we statically analyze to guarantee memory safety. We essentially propose a combination of static analysis, minimal (often zero) run-time checking and some system support (only to minimize the run-time overhead) to ensure that uninitialized pointers, type-unsafe casts, dangling pointers to freed memory or stack memory, and array bounds violation do not overwrite any data area not allocated by the component. In our case, since shared memory is allocated through the shared memory libraries, our analysis and language restrictions enforce that local pointers in the core component cannot access any shared memory locations. In a follow-up work [31], we describe the additional run-time checks required to guarantee memory safety (in fact, the stronger property of guaranteeing the semantics of the pointer analysis results) for practically the full generality of C.

The second consequence of type-unsafe constructs is imprecision in tracking shared memory location accesses leading to a greater number of false positives in a conservative analysis. To statically analyze value flow precisely, we impose a few semantic restrictions on shared memory pointer usage. This enables better precision , ensures expressiveness to program practical embedded systems, and handles legacy systems with minimal porting effort.

We enforce that a pointer to shared memory cannot be aliased and arrays in shared memory cannot be indexed. Without such restriction, it would be impossible to statically analyze the component. Also, we require that shared memory is not deallocated or destroyed until the end of the program (the end of function `main`). This prevents dangling pointers to shared memory in the program. Dereferencing dangling pointers to system-defined shared memory can have unpredictable consequences such as crash failures of the core component. The restrictions are as follows: (**P1**) Shared memory cannot be deallocated until the end of the main function; (**P2**) Taking the address of a pointer to shared memory is disallowed; and (**P3**) Casts between pointers to incompatible types in shared memory and casts from shared memory pointers to integers is disallowed.

In addition , we adapt the restrictions on arrays in our previous work [33] to the arrays in shared memory:

(**A1**) Indices used to access arrays within shared memory must lie within the bounds of the array

(**A2**) If an array in shared memory, `A`, is accessed inside a loop, then: **(a)** the bounds of the loop must be provably affine transformations of the size of `A` and outer loop index variables or vice versa;**(b)** the index expres-

sion in the array reference, must be a provably affine transformation of the vector of loop index variables, or an affine transformation of the size of A; and**(c)** if the index expression in the array reference depends on a symbolic variable S, which is independent of the loop index variable (i.e., appears in the constant term in the affine representation), then the memory locations accessed by that reference have to be provably independent of the value of S.

Rule **P1** above prevents dangling pointers to shared memory. Rule **P2** disallows aliasing shared memory pointers by storing them in memory. **P3** ensures that the shared memory is used in a type-safe manner. The array rules **A1** and **A2** verify that the arrays in shared memory do not violate their bounds. In general, inability to distinguish statically between array locations results in an array index operation conservatively assumed to be anywhere within the span of the memory locations represented by the array.

```
  ...
  initComm(key_t SHMKEY, size_t SHMSize,
           FLAGS flags)
  /***SafeFlow Annotation
      assume(shminit); ***/
  {
1:   void *shmStart;
     /* Initialize shared memory */
2:   shmid = shmget(SHMKEY, SHMSize, flags);
3:   shmStart = shmat(shmid, 0, 0);
4:   feedback = (SHMData *) shmStart;
5:   noncoreCtrl = feedback + 1;
     /***SafeFlow Annotation
         assume(shmvar(feedback,
                       sizeof(SHMData))) ;
         assume(noncore(feedback));
         assume(shmvar(noncoreCtrl,
                       sizeof(SHMData)));
         assume(noncore(noncoreCtrl));
     ***/
     InitCheck(shmStart, SHMSize,
               feedback, sizeof(SHMData),
               noncoreCtrl, sizeof(SHMData));
  }
  main()
  {
1:   initComm(SHMKEY, SHMSize, flags);
2:   Lock(shmLock);
     ...
```

Figure 3.3: Initialization function

Shared Memory Initialization

In addition to the `assume` and `assert` annotations described in section 3.2.1, we require one other annotation in order to facilitate our analysis. This is due to shared memory allocation through systems calls (in UNIX for instance) being untyped, which necessitates shared memory pointer casting

and pointer arithmetic in order to initialize shared memory. These constructs violate the restrictions on shared memory pointer usage described above. Moreover, due to the type-unsafe constructs, the sizes of arrays in shared memory need to be made explicit. We also require annotations to specially designate that these initialization routines do not require to adhere to our language restrictions.

For instance, lines 1-5 of `main` in figure 3.2, the pointer returned by the call to `shmat` is cast to a pointer to a structure of the returned type (violating rules **P3**). In order to overcome this, one option is to write and invoke typed wrappers for these system calls for each required type. This is clearly onerous on the developer. In our approach, we designate that the initializations are performed in a special function known as the *initializing function*, identified by the `assume` annotation using the predicate `shminit`. The predicate, `shminit`, permits **P3** to be violated.

To overcome the type-unsafe initialization of shared memory, the initializing function needs to be annotated to identify the shared memory variables and their respective sizes. For this purpose, we employ `assume` annotations on the predicate `shmvar`, which takes the typed shared memory pointer as the first argument and the total size of the shared memory locations that can be accessed through the pointer as the second argument:
`shmvar(shmptr, size)`
The size of the array pointed to by the shared memory pointer can be inferred by dividing the size of the shared memory, `size`, by the size of the type pointed to by `shmptr`. It is important to verify that the locations pointed to by individual shared memory pointers are non-overlapping and the entire size span of each of these pointers be valid. To annotate that the set of locations that can be accessed by a shared memory pointer are non-core, we employ the `assume` annotation on the predicate, `noncore`, if the shared memory locations can potentially be overwritten by a non-core value. This predicate is applied to a shared memory pointer as follows:
`noncore(shmptr)`

In figure 3.3, we show an annotated version of `initComm` with all the above annotations. The `shminit` annotation is written just below the function declaration and applies to the function and any function invoked recursively by it. The `shmvar` and the `noncore` annotations are written at the end of the function and are post-conditions of the initializing function. In this case, `feedback` and `noncoreCtrl` are declared to be two shared memory variables of size `sizeof(SHMData)`. In order to assist the programmer in writing correct size annotations, we automatically insert a run-time check:
`InitCheck(void *SHMStart, size_t SHMSize, ...)`
which verifies that the variables in shared memory do not overlap with each other. If this check fails, the core component is terminated before it

bootstraps. While this is a run-time check, it is only executed once during shared memory initialization.

3.2.3 Static Analysis

Our static analysis algorithm operates in three phases on the core component implementation: (i) Identification of pointers to shared memory interprocedurally; (ii) Enforcing language restrictions **P1**-**P4**, **A1**, and **A2**; (iii) Identifying non-core shared memory accesses and determine if critical data is control or data dependent on unsafe local values. The analysis is implemented on low-level virtual machine (LLVM) byte-code [63], which is a typed intermediate format that is in static single assignment (SSA) form.

Before the three phases of our analysis, we execute a pre-processing pass on the C code which converts `assume` and `assert` annotations to calls to external dummy functions. Annotations on monitoring functions and initializing functions are specified at the entry point of the function. The post-condition of the initializing function is generated at all the exit points of the function. In the first phase, we discover the initializing functions in the program and identify the shared memory pointers initialized. We then propagate these pointers interprocedurally using a bottom-up and top-down analysis on the strongly connected components (SCCs) of the call graph. These pointers escape the functions as globals, function arguments or return values. The post-conditions of initialization functions identify the pointers to shared memory that escape the function. In the bottom-up pass, these pointers are propagated to the callers until the root of the call graph i.e. `main` is reached. Within each function, a standard global data flow algorithm is used on the basic blocks in the control flow graph (CFG). At merge points (due to conditionals or loops), a pointer is conservatively assumed to point to shared memory if it is initialized on some path. Within SCC's, the pointers to shared memory are propagated to the callers until the shared memory pointer information for each function stabilizes. A top-down pass on the call graph propagates the pointers to shared memory from the root of the call graph to the callees.

The second phase of our analysis enforces the language restrictions **P1**-**P3**, **A1**, and **A2**. According to rule **P1** pointers to shared memory in each function should not be arguments to shared memory deallocation functions (`shmdt` for instance). This is easy to analyze by examining all the uses of the pointer by following def-use chains. **P2** and **P3** can be similarly verified by examining the uses of pointers to shared memory within each function. Casting between types in LLVM necessarily uses the `cast` instruction. Casting a pointer to shared memory to a pointer to an incompatible type is disallowed (**P3**). **P2** is verified by enforcing that a pointer to shared memory is never stored in any pointer using the `store` instruction

in LLVM.

Our restrictions on array indexing are derived from a previous work [33]. Constraints **A1** and **A2** are verified by generating constraints on each array index expression in the program interprocedurally. The constraint propagation algorithm is exactly the same as the one developed previously. The size of the array in shared memory is provided by the annotation in the initializing function. The set of affine constraints are given to a integer programming solver such as Omega [57], which checks that there are no array bounds violations.

The final phase of our algorithm processes the `assume` annotations in each function (except the initialization function) and determines the core set of shared memory locations accessible within each function. This is compared to the non-core shared memory locations accessed in the function to determine the unsafe values read from unmonitored shared memory. A warning is reported for each unsafe access to shared memory, without any false positives or false negatives. Finally, we enforce that critical data is not data or control dependent on unsafe shared memory accesses using an interprocedural value flow analysis on the critical data. An error is reported when the analysis detects dependency of critical data value in the core component on unmonitored non-core values.

In order to accomplish this, we propagate the predicate, `unsafe`, which is applied on pointers and primitive data types loaded from unmonitored non-core shared memory using the rules below. Note that, in general, a pointer is unsafe if and only if the data it points to is unsafe.

1. `Load` : Any value loaded from an unsafe pointer is considered unsafe.
   ```
   q = load p
   unsafe(p) ⇒ unsafe(q)
   ```

2. `Arithmetic operations`: Any data value computed from an unsafe value is unsafe.
   ```
   c = add a, b
   unsafe(a) ∨ unsafe(b) ⇒ unsafe(c)
   ```

3. `Branch Merges (Phi nodes in SSA)`: If a variable is unsafe on any path, then conservatively this unsafe predicate is propagated on merging paths.
   ```
   c = phi(a,b)
   unsafe(a) ∨ unsafe(b) ⇒ unsafe(c)
   ```

4. `Conditionals`: If a variable that affects a conditional is unsafe, then all value definitions that are control dependent on the conditional are unsafe
   ```
   if (cond) x = y
   unsafe(cond) ⇒ unsafe(x)
   ```

5. `Function call`: Unsafe data or pointers are propagated to and from functions through arguments, return values, and globals.

6. `External libraries`: If unsafe data or pointers are propagated to an external library (whose source is unavailable), then all return values, accessible globals, and pass-by-reference arguments are considered unsafe.

We verify the assertion for critical functionality, by checking if the critical data depends on an unsafe value. We use an alias analysis like Data Structure Analysis (DSA) [62], which maintains points-to graph and a typed representation of the memory in the program. DSA is a context-sensitive, field-sensitive, and flow-insensitive analysis.

The critical data analysis algorithm checks for dependencies on unsafe data read by the core component from shared memory using an interprocedural, context-sensitive, and flow-sensitive algorithm. Currently, each function in the core component is analyzed multiple times for different call sequences leading to it, making the implementation exponential in runtime complexity. In practice, the core component in an embedded system is simple and has relatively fewer paths than system software. Moreover, the overhead due to static analysis time for safety violation detection is not a significant factor in most development and testing efforts.

In the example in figure 3.2, the shared memory pointer, `feedback`, is used in the function, `decision` (which only annotates `noncoreCtrl` as being safe). Thus, any values generated by `decision`, which depend on `feedback` are unsafe. This includes the return value, `output`, which violates the critical functionality requirement of the component. The dereferencing of `feedback` in `decision` is reported as unsafe. One way to eliminate this dependency is to use a local copy of the feedback as an argument to `decision`, rather than the pointer to the shared location.

The algorithm can be made more efficient by analyzing each function only once and summarizing the data dependencies in the functions using *value flow graphs* developed in ESP [27]. This ability to summarize procedures means that we can carry out a single bottom-up pass on the SCC's in the call graph, inlining the value flow graphs in the callers and using these graphs to determine if critical data depended on any unsafe accesses to shared memory.

3.2.4 Discussion and Extensions

False Positives

Our analysis detects false positives, due to two reasons: the imprecision in our analysis and control dependence on shared memory variables. The merging of unsafe predicates on variables upon branch merges and the

path-insensitivity in the final phase of our analysis can result in false positives, due to errors flagged by infeasible paths. In the future, our analysis could be combined with path-sensitive value flow algorithms such as ESP [27]. Similarly, the imprecision of the pointer analysis used to check if any unsafe data is reachable from critical pointer data can also lead to false positives. While improving the precision of the pointer analysis results using more aggressive analyses (e.g. making it flow-sensitive) increases precision, eliminating false positives in all cases is difficult.

The second source of false positives is due to critical data being control dependent on unmonitored non-core shared memory values. For instance, in one of our test-cases, the configuration of the system is present in shared memory. The core component reads the configuration without monitoring it and computes critical data differently based on the presence or absence of a non-core component. In one path of execution, the critical data uses the non-core values after monitoring it. In the other path of execution, the core component outputs safe data computed by it. The critical data is computed correctly in either path of execution, but the control dependence on the non-core configuration data reports an erroneous dependency. The source of this false positive is due to the inability of the analysis to automatically infer whether the non-core variables modifying the control flow affect the critical data computation in the core component. In these cases, manual inspection of the reported errors is required. However, it is important to realize that in these scenarios, a superior design would be to restructure the non-core components by separating out an additional core component that writes the configuration in shared memory.

Non-core component encapsulation

Due to our conservative model of the non-core component, the core component cannot rely on atomicity properties such as writing and reading a shared variable in sequence and expecting the written value to be read. In Earlier, we motivated our conservative model of the values produced by the non-core component as being due to the difficulty of detecting many complex errors in non-core component behavior. In special cases, more fine-grained model of non-core component behavior could be exploited, due to guaranteed absence of errors such as synchronization errors or data compatibility errors. This can easily be expressed by using more `assume` annotations in the core component to declare shared memory locations to be `core` within certain functions. For instance, in the example in figure 3.2, the function `decision` could be further annotated with `assume(core(feedback, 0, sizeof(SHMData)))`, thus declaring feedback to be safe to dereference in `decision` and all the functions recursively called by it. A limitation of this approach is that annotations in our current approach can only be applied at

the function level and might necessitate restructuring the code to further modularize the program appropriately.

Message Passing and I/O calls

In this section, we briefly discuss extending our approach above to communication through message passing and I/O library calls. We define a trusted set of library calls we can use for receiving messages and performing I/O reads. We illustrate the required annotations on the library calls using the example of the `recv` call on sockets:

```
ssize_t recv(int socket, void *buffer,
             size_t length, int flags);
```

First, we use the predicate, `noncore(socket)`, to specify that socket file descriptor, `socket`, is used to communicate with non-core components. Otherwise, the socket is assumed to be used for communication with core components. Additionally, we require that these descriptors are not used in any computation and merely passed by value across procedures. Socket file descriptors not annotated as non-core are assumed to communicate with core components. In practice, the sockets communicating with core components need to contain run-time authentication to check that the peer components are indeed a part of the core subsystem. Secondly, we use `assume` annotations to define that it is safe to dereference received non-core data within the function. This is exactly the same as the `assume(core(...))` annotations in the monitoring functions, except that it is applied to a local pointer. All other pointers to received data are assumed to be unsafe, being from non-core components.

In the example below, `noncore` is deemed safe to dereference, but not `feedback`. To facilitate analysis, we require that the address of pointers to received buffers are not taken and these pointers are not cast to integers. An annotated snippet of a version of the `decision` function in our running example implemented using message `recv` is shown below:

```
float decision(float safeControl)
/***SafeFlowAnnot:
    assume(safe(noncore, 0,
                sizeof(SHMData))); ***/
  {
1:   n = recv(socket, buffer, length, flags);
2:   feedback = (SHMData *) buffer;
3:   noncore = feedback + 1;
4:   if (checkSafety(feedback, noncore))
5:      return noncore.control;
6:   else
7:      return safeControl;
  }
```

3.3 Results

Through our experiments, we intend to answer the following questions:

- Is the restricted language expressive enough to develop embedded control systems?
- Are the annotations onerous on the programmer?
- Does the analysis successfully detect erroneous dependencies on non-core values in our tests?

System	LOC (total)	LOC (core)	Source Changes (LOC)	Annot. lines count
IP	7079	820	7 (86)(1 func)	11
Generic Simplex	8057	1020	0	22
Double IP	>7188	929	7 (88)(1 func)	23

Table 3.1: SafeFlow: Programmer Burden Evaluation

System	Error Dependencies	Warnings	False Positives
IP	1	7	2
Generic Simplex	2	7	6
Double IP	2	8	2

Table 3.2: SafeFlow: Error/Warning Detection Results

Table 3.1 and 3.2 shows the results of applying our analysis on three laboratory control systems. All three systems implement the robust architectural design of isolating the core components and monitoring for non-core values flowing to the core component. The first system is a Simplex architecture for an inverted pendulum (IP) controller used to balance an inverted pendulum as seen in our running example (figure 3.1). The second is a generic Simplex architecture implementation for simple plants with a configuration file that can be customized for different plants. The third system is a double inverted pendulum control system that is based on the inverted pendulum controller code, albeit with changes to enable additional control modes. The first two systems have been used in the real-time systems laboratory at UIUC for three years and have been extensively tested. In particular, these systems were designed as a demonstration of the Simplex architecture for core component isolation. Much effort has gone into the development of these two systems, particularly in protecting against non-core values in the core component. The double IP controller is a relatively new system, whose implementation is currently being refined in our laboratory. For running our analysis, we used a preliminary version of the double IP controller.

In order to apply SafeFlow, we needed to annotate the core components of the systems with information about shared memory initialization. The critical data in the core components is the control output being sent to the actuator and the first argument (process id) of the `kill` system call invoked (annotated using `asserts`). Finally, we required annotations on the monitoring function specifying the non-core accesses that are safe to access within the function. The number of lines of annotation is small in all cases. In particular, majority of the annotations (9 of 11 lines in IP control, 15 of 22 in generic Simplex, and 15 of 23 in double IP control) were used to annotate initializing functions. Notably, no source changes were necessary for the systems to adhere to our language restrictions. A very small number of source changes were required in two experiments to separate the monitoring function, which was a part of a larger function. This was necessary because the annotations for the monitoring functions can only be specified at the function level. In table 3.1, the number of actual lines of source changed and the `diff` output of the modified program with respect to the original program are listed.

The SafeFlow analysis detected several warnings or unmonitored non-core value accesses in all three systems. These warnings contain no false positives are a very useful output of the analysis since it makes explicit all unmonitored non-core values read by the core component. SafeFlow detected two erroneous value dependencies of the core component on the non-core component in the generic Simplex system, two in the double IP system and one in the inverted pendulum controller. In the generic Simplex implementation, one erroneous dependency was caused by a shared memory variable (the sensor feedback value) being written by the core component and read later by the core component. This potential value dependency on non-core values would be fatal, if the non-core component replaced the sensor feedback with a hand-crafted value that would "rig" the recoverability check to permit an erroneous non-core value to be used by the core controller. This could occur due to an erroneous non-core component implementation, which overwrites the feedback value (which is supposedly read-only, but not enforced) or violation of the synchronization on the feedback value in shared memory due to data races.

In all the three systems, the first argument of a `kill` system call invoked by the core component was dependent on an unmonitored non-core value. This could be easily used to bring down the core component if the non-core component overwrote the value with the process id of the core component itself, causing the core component to kill itself! One error in the double IP controller is a result of accessing an unmonitored non-core value assuming that this value does not propagate to the critical data in the core component. Our analysis discovers that this assumption is invalid. It is important to note that the three systems tested were designed and imple-

mented to provide safe value flow from non-core to core components. The five errors we found are very subtle, in that they capture implementation oversight and erroneous assumptions.

SafeFlow returns a few false positives among its errors. All false positives returned in our tests were due to control dependence on non-core values that do not affect critical data computation. These needed to be manually identified with the aid of the value flow graphs representing the flow of values from unmonitored non-core values to the critical data. False positives can be reduced by using the `assume` annotation to declare such non-core values as being safe to access within certain functions, only after reliably verifying this fact.

3.4 Related Work

As described earlier, there have been several architectural designs that employ monitors to protect core components against non-core values [82, 26, 46]. The notion of isolating critical component functionality has been formally modeled by Arora and Kulkarni [5] in order to build masking fault-tolerant systems. Further, Jhumka *et al* [54] have proposed the design of accurate and complete detectors in such an architectural design. However, none of these works have addressed enforcing these principles in low-level implementation and have verified their designs only at the model level, which is a much higher level of abstraction. We have shown that implementation errors can easily violate even the simple architectural principle of safe value flow.

The SafeFlow approach is closest to previous work in taintedness analysis and secure value flow. The taintperl [93] package employs data flow techniques to quarantine data potentially contaminated by malicious users (completely preventing the use of such data). Being an interpreted language, the taintedness information is tracked at run time. In the context of security, Denning and Denning [29] first proposed a type system enhanced with security attributes in order to verify confidentiality and integrity of variables in the implementation. This has been further refined by Volpano *et al* [90] and in the design of JIF [71, 80]. None of these works are suited for the safe value flow property in embedded systems, for two main reasons: heavy-weight annotations and the lack of a notion of monitoring functions. Moreover, prior work in security typed languages has not practically addressed applying the techniques to C programs. In contrast, the key contribution of SafeFlow is the design of a succinct and light-weight annotation language to specify monitoring function properties and language restrictions that enables statically detecting erroneous dependencies. While secure programming languages can be justified to guarantee

confidentiality and integrity in security-critical applications, safe value flow demands legacy code compatibility and ease of use by embedded system developers. Declassification in JIF is the closest analogue to monitoring in literature, although the semantics are different. We borrow our value flow graphs to propagate the `unsafe` predicate on a local variables through the program from the propagation of security attributes towards guaranteeing non-interference, for instance. This is also similar to Cqual [84] which contains mechanisms to annotate interface variables with attributes and propagate these attributes through the program (and potentially to assertion or interface violations).

A key element of our approach is that the reading of non-core values is automatically detectable by explicitly annotating shared memory initialization, and message receive and I/O library calls. The Ada-SPARK system [8] utilizes annotations to make explicit the data flow information in a program, thus avoiding implicit value propagation errors for a subset of Ada. Their system was intended at making the behavior of individual functions deterministic and analyzable. Enforcing the guaranteed containment of propagated value errors from non-critical components has not been addressed from the point of view of dependability. In fact, using Ada's explicit value propagation approach would be useful in eliminating many false positives in the final phase of our analysis, where we report the critical values that depend on unmonitored non-core values.

Leveson [69] has developed a manual technique to analyze lines of code to generate software fault trees for each line of an Ada program. The manual technique and the large size of the fine-grained trees preclude the scalability of the technique. Automatic failure path inference (AFPI) [15] attempts to determine failure dependencies across components through fault injection and testing methodology, without offering any guarantees

Enforcing locking protocols and detecting data races statically has been studied extensively, most recently in [35]. Data format compatibility and access control can be enforced by encapsulating the shared memory reads and writes by the non-core component and through assumption specifications in AADL [39]. These techniques are best-effort and do not attempt to capture all errors. Thus, assuming the absence of such errors is unjustified in general, motivating SafeFlow to use the conservative model for the non-core component in its analyses.

3.5 Chapter Summary and Future work

We described SafeFlow, an annotation-based static analysis that verifies that the core components guaranteeing critical functionality do not depend on unmonitored non-core values in the system. While monitoring has been

used extensively in robust architectural design, various implementation errors can violate these architectural principles. The analysis tool enables core component developers to explicitly track non-core values in shared memory that are used without being monitored in core components. Further, the tool reports erroneous dependencies if these non-core values can affect critical data computation along some path in the system. Applying SafeFlow found critical, erroneous or inadvertent value dependencies in mature critical systems designed for safe value flow, with a few false positives. The SafeFlow experiments confirm the motivation behind the analysis tool as being a "final line of defense", complementing other best-effort error detection tools.

In the future, SafeFlow needs to address other channels of value flow between non-core and core components, significantly *storage channels*. Further attention is required on eliminating the false positives generated by SafeFlow.

Chapter 4

Developing Robust Embedded Systems

In this chapter, we briefly summarize the contributions of chapter 2 and chapter 3 and put these contributions in perspective by analyzing the extent of our achievement towards the broader goal of enabling the system integration architect to verify architecture-implementation conformance. We also examine other related efforts and some simple extensions, which are potentially useful to general-purpose systems or embedded systems that cannot adhere to the restrictions imposed by Control-C or SafeFlow.

We embarked on this work in order to bridge the gap between the implementation and robust architectural assumptions or requirements. In order to do so, we chose a key emerging architectural principle for robustness, which guarantees critical functionality even in the event of faults in the non-critical subsystem, the separation principle. The separation principle comprised three sub-properties: (SP1) spatial separation, (SP2) temporal separation, and (SP3) safe inter-component interaction. There are a multitude of failures at the implementation level that can violate each of the above properties. Among the wide spectrum of potential solutions including formal analysis, dynamic monitoring, static analysis, and system utilities, we chose a solution based on static analysis with minimal support from system utilities, dynamic checks, and programmer assistance.

Static analysis for architecture-implementation conformance is employed by the system integration architect to validate the implementation. In our analysis, static analysis is directly used by individual developers to analyze their code: (a) non-core components and core components are coded in Control-C to guarantee memory sandboxing and no hidden system calls, (b) the system calls invoked by non-core components is restricted to a safe subset determined by the system integration architect, and (c) the core components are annotated by the developer and verified by SafeFlow to run-time monitor all non-core values read from shared memory. The system integration architect simply needs to ensure that the components code by the individual developers have been analyzed by our tools. This can be simply enforced by modifying the build scripts, such as the `makefile`. In addition, the system integration architect needs to enforce that assumptions made on the run-time system and on the correctness of run-time monitors are

valid.

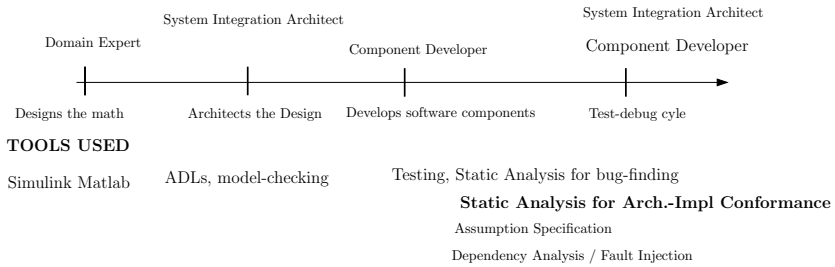


Figure 4.1: Embedded system development timeline

Figure 4.1 demonstrates the relative position of our tool compared to the other tools available for system development. The timeline shows the following phases:

1. The domain expert employs prototyping tools such as Simulink and Matlab. In a traditional control system, these prototyping tools could be used to directly program the software components.

2. The system integration architect designs the system to satisfy the functional requirements (as solved by the domain expert), and, in addition, also provide non-functional features such as dependability and security. The system integrator can use architecture description languages to validate his designs at the modular level. In addition, she also determines the run-time system utilities required by the individual components.

3. The individual software developers program their components according to the specifications established by the architect. In doing so, they employ testing and debugging tools (particularly static analysis tools which try to find common programming errors).

4. The system integration architect verifies that our static analyses for architecture-implementation conformance have analyzed the core and the non-core components in the system. In this stage, the system architect and the individual developers also evolves the assumption specifications [87], the timing specifications, and performs the failure dependency analysis on the system.

4.1 Our Approach

Leveraging static analysis has tremendous benefits for embedded systems, viz., run-time, memory, and power performance, and early error detection. Our approach towards enforcing high-level properties at the implementation-level through static analysis is distinct from many alternate approaches in

literature that employ static analysis. In particular, other work has attempted to use static analysis and/or run-time checking to identify programmer errors wherever possible, thus providing early feedback to the programmers in the development cycle. These approaches are "best-effort" in nature and their main focus is in minimizing the number of false positives reported. The tools continue to generate false negatives in the form of bugs that are difficult or impossible to detect by these tools. Our approach atttempts to provide architectural guarantees at the implementation level, particularly towards the high-level properties related to guaranteeing critical system functionality in the presence of arbitrary errors in the non-critical subsystem.

Our approach primarily consists of two key steps:

- First, the high-level property to be enforced is carefully selected, keeping in mind that it needs to be *guaranteed* at the implementation level statically, with minimal or no run-time checks.
- Using a combination of minimal language restrictions and annotations, sophisticated compiler analyses, judicious run-time checking, and modifications of the run-time environment to statically guarantee that the high-level property identified in the previous step.

The first step above has proved to be crucial in both Control-C as well as SafeFlow and is a key insight of this work. We illustrate this in both cases. Detecting all dangling pointer errors in C is basically an intractable problem. Thus, in developing Control-C and the accompanying static analyses, we modify the heap allocation and deallocation routines to allocate many fine-grained heaps, each of which are type homogenous. In this way, we can exploit the type-homogeneity principle from section 2.3.3 to guarantee that memory safety is guaranteed in the presence of dangling pointer dereferences. While verifying strong type safety is sufficient in guaranteeing spatial isolation, it is not necessary. Thus, we permit dangling pointer dereferences in some cases (thus violating strong type safety of the program in these cases), but ensure that these violations do not compromise memory safety.

In SafeFlow, we enforce that erroneous values from the non-core subsystem are run-time monitored by the core components before being used. There are typically various difficult-to-detect bugs and incompatibilities in the interaction between the core and non-core subsystem that can violate the safety of the interaction. For instance, statically verifying the correctness of the locking protocol to access the shared memory is difficult if not impossible. Similarly, data format compatibility between the critical and the non-critical subsystem in the message or the shared memory is tedious to guarantee, particularly when the interface evolves over time. By using

static analysis to verify that *all* values read from shared memory are monitored before use, we capture the effect of any arbitrary interaction error resulting in bad data.

By carefully choosing the property we require to guarantee that the implementation adheres to the architectural design assumptions, we can enforce that our language restrictions are minimal and retains the expressiveness of the language and that we incur minimal (often zero) run-time overhead, through explicitly inserted run-time checks or due to run-time system utilities.

4.2 Case Study: Re-designing eSimplex

By applying the techniques described in chapter 2 and 3, we have analyzed the design of eSimplex for simple control systems including the inverted pendulum. The results of these experiments have showed us the following

1. The Simplex implementation adheres to the restrictions of Control-C including the type-safe pointer usage and affine array indexing. Also, it runs on a system (embedded Linux) which contains a reserved address space eliminating run-time software checks for uninitialized pointers. Thus, eSimplex was completely statically verified for memory isolation, with minimal or no source changes. Our analysis also enforces that there are no hidden system calls in the data area within the component.

2. The implementation contains unsafe value propagation paths from the non-core controller to the core controller. In particular, we found two critical bugs in both these systems:

   (a) A hidden dependency of the safety controller implementation on the feedback tuple in shared memory that can be overwriten by the non-core controller. This is either an inadvertent dependency or a synchronization assumption on the part of the programmer that can be easily violated by the non-core controller. In particular, the feedback value can be overwritten with arbitrary values, which can be used to "rig" the safety check. To fix this problem, the implementation needs to be modified to use the local variable containing the feedback tuple read from a core sensor, rather than the non-core value in shared memory.

   (b) The SafeFlow analysis inferred that the first argument of the `kill` system call (process id) in the core component depends on a shared memory value written by the non-core component. The non-core component is expected to register its own process id in

this shared memory location. This can be overwritten by the non-core component that leads to the core component killing itself or other core components. The process id to be killed either needs to be monitored by the core component or a core component needs to be separated out of the non-core component, which registers the process id of the non-core component and monitors it for safety.

Finally, the SafeFlow analyses showed a control dependence of the critical control value in the safe controller on the values in shared memory that indicate the configuration of the system. This configuration indicates the presence or absence of a non-core controllers in the system. Currently, these parameters are written by the non-core controller. The dependency detected in the core component is a false positive since the core component generates critical values correctly irrespective of the configuration. However, a better design of the system to avoid any future potential erroneous dependencies on the configuration would be to separate out a core component from the non-core component, which writes the configuration of the system in shared memory. This component would monitor the configuration of the non-core subsystem and only writes reliable (core) configuration values to shared memory.

4.3 Extensions to Control-C and SafeFlow

The language restrictions on Control-C and the annotations in SafeFlow are the two most significant bottlenecks to widespread deployment. In this section we discuss extensions and approaches that can eliminate these restrictions, at the expense of additional run-time overheads or programmer assistance.

4.3.1 Extending Memory Safety to C

Control-C was developed as a part of the SAFECode project at UIUC (see `http://safecode.cs.uiuc.edu`). Extending memory safety to the full generality of C is a challenging task and is not a contribution of this dissertation. It has been described by Dhurjati *et al* in [31, 30]. We summarize the key results here.

In order to support the full generality of C, we need to relax two significant restrictions in Control-C

- Casting between any types including incompatible pointers and pointers-to-integers and vice-versa.
- Allowing complex array indexing, including using values loaded from the heap as indices and non-affine indices.

Enabling the above semantic features in C necessitates adding some run-time checks in order to guarantee memory safety without garbage collection. In particular, there are two categories of run-time checks thare are necessary. First, run-time checks are used to verify the safety of incompatible type-casts and for array indexing that cannot be statically proved to be safe. Broadly, these set of run-time checks verify that a pointer does not point outside memory allocated to it. The reason for the second category of run-time checks is subtler in nature. In performing static analysis for validating certain array acccesses and prevention of dangling stack pointers, we utilize the pointer analysis results and the computed call graph of the program. However, in the presence of type-unsafe language constructs, the pointer analysis information and the call graph can become invalid or erroneous. This compromises the correctness of our static analyses. In order to enforce that our static analyses are correct, we need a second category of run-time checks that enforce the call graph and the pointer analysis results.

In fact, our run-time checks for memory safety additionally guarantees the semantics of pointer analysis results and call graph results for any analyses that rely on this information. Enforcing sound alias analysis results through run-time checks (in turn, optimized using static analyses) is useful for a large number of static analysis tools such as BLAST [50] and ESP [27].

In implementing these run-time checks, the goal is to incur minimal run-time overhead. In order to execute an efficient run-time check, we segregate the memory used in the program to distinct heaps and distinguish between type-homogenous pools (such as the all the pools in Control-C) and type-unknown pools for memory used in a type-unsafe manner. We convert all run-time checks into efficient pool checks. The resulting overhead incurred was less than 10% in most of the cases and a maximum of 30% (in one case).

4.3.2 Safe Shared Memory Library

The SafeFlow analysis combines programmer annotations with restrictions on using pointers to shared memory. While our simple and local programmer annotations enabled us to handle C codes, future systems can potentially employ a more elegant technique based on a safe set of library calls in accessing shared memory.

We define the following library routines and illustrate their use with figure 4.2:

```
void *shminit(key_t key, size_t, size, int flags)
void shmdt(Void *SHM)

bool register_monitor(void *SHM,
```

```
      bool (*monitorFunc)(char c, void *MFArgs), size_t offset)
bool register_monitor(void *SHM,
      bool (*monitorFunc)(int i, void *MFArgs), size_t offset)
bool register_monitor(void *SHM,
      bool (*monitorFunc)(float f, void *MFArgs), size_t offset)

bool writeSHMVal(void *SHM, size_t offset, char val)
bool writeSHMVal(void *SHM, size_t offset, int val)
bool writeSHMVal(void *SHM, size_t offset, float val)

bool readSHMVal(void *SHM, size_t offset, int *val,
                void *MFArgs)
bool readSHMVal(void *SHM, size_t offset, int *val,
                void *MFArgs)
bool readSHMVal(void *SHM, size_t offset, float *val,
                void *MFArgs)
```

The library function `shminit` is used to initialize an untyped shared memory of a given size, associated with a key, and flags indicating permissions. Run-time monitors can be registered for elements within the shared memory, which are identified by an offset from the beginning of the shared memory. For this purpose, `register_monitor` is overloaded to be used to monitor each distinct data type (in the library routines above, we have restricted ourselves to `char`, `int`, and `float`). Monitoring functions take in the value to be monitored as the first argument and a void pointer as the second argument. The void pointer enables monitoring function to use custom structures of the inputs used in monitoring. For instance, a monitoring function for a control value could use a structure with a single floating point feedback value or an array with a window of feedback values. In figure 4.2, the run-time monitor, `valMonitor`, monitors the control value, `ctrl`, by using the floating point feedback value as the second argument. Monitoring functions return true if the value is safe and false otherwise.

We do not allow pointers to be explicitly stored in shared memory. The routine, `writeSHMVal`, can store values in shared memory at a given offset. The routine `readSHMVal` is more complex. It attempts to read the value, `val`, of a certain size from a given offset in shared memory. In doing so, it first checks if there is a monitoring function for the value of this type at the given offset and returns false if none exist. If there is a monitoring function, `readSHMVal` invokes the monitoring function. The function, `readSHMVal`, returns false if the argument `MFArgs` is null or if the invoked monitoring function returns false. Otherwise, it returns true. The read value is made available in $*$`val`. The monitoring function is relied upon to verify that the arguments passed in as `MFArgs` is valid. Also, each of `register_monitor`, `writeSHMVal`, and `readSHMVal` verify that the offset is within the bounds of the shared memory.

```
bool valMonitor(float ctrl, void *Args){
  float *feedback = (float *) Args;
  ...
  // Returns true if ctrl is safe
  // i.e. ctrl retains the system in a recoverable state
  // else returns false
}

main() {
  SHMStruct *theSHM;
  float feedback = readFeedback();
  theSHM = (SHMStruct *) shminit(SHMKey, sizeof(SHMStruct),
                                 FLAGS);
  register_monitor((void *) theSHM, &valMonitor,
                   sizeof(theSHM->fld0));

  float ctrl;
  if (readSHMVal((void *) the SHM, sizeof(theSHM->fld0),
      &ctrl, (void *) &feedback)) {
     // ctrl is safe
  } else {
    ctrl = SAFE_VALUE;
  }
  sendControl(ctrl);
}
```

Figure 4.2: Example: Using the safe shared memory library

Using the safe shared memory library, the static analysis required to enforce that all shared memory values are run-time monitored before use in a critical operation in the core component reduces to a few simple rules:

1. The shared memory pointer is only used as the first argument of the safe shared memory library calls and conversely, the first argument of the safe shared memory library calls should be the return value of `shminit`.

2. The function `shmdt` should be invoked on the shared memory pointer only at the end of `main`.

3. The return value of `readSHMVal` is always checked.

4. The value pointed to by `val` is safe along the path where `readSHMVal` returns true and unsafe along the path where it returns false.

A simple analysis identifies the unsafe values in the core component and a value flow analysis similar to the one described in chapter 3 is used to verify that unsafe values do not propagate to critical data in the core component (identified by the same `assert` annotation used in chapter 3. Converting existing programs to use the safe shared memory library involves major program restructuring (More than 300 lines of changes in our systems, where the core component had 800-900 lines). Hence, the library

would be most effectively used in systems to be developed in the future and not for legacy codes.

4.4 Related Work and Future Directions

We motivated the importance and utility of statically identifying implementation issues that violate architectural precepts. The closest work that adheres to these goals are language-level solutions. Java [45] guarantees memory safety and spatial isolation. Additionally, there have been several language level solutions to use a combination of annotations and static inferencing to discover erroneous value flow dependencies, particularly in the context of secure information flow. JIF [80] contains language extensions to Java, which guarantee properties such as integrity, confidentiality, and non-interference. The emphasis of JIF has been on a usable language, making it unique in a long line of work in this area. Java has not been used extensively in embedded system development because of the unpredictable nature of garbage colleciton and the run-time checking overhead. More recently, there have been successful efforts at practically enabling real-time garbage collection [74].

There are a few attractive directions for future work to pursue. We have addressed single-threaded programs in this work. Extending the architectural properties to multi-threaded codes is still an open challenge. Also, the techniques used in SafeFlow and the safe shared memory library can be extended to verifying many other architectural properties such as security and appropriate filtering of data to guarantee timely data. One such architecture for filtering feedback to negate the effect of delays or inaccuracies is proposed in appendix B.

Specifically, with regards to the analyses in this work, three distinct improvements can be envisioned. First, our ability to verify safe system call usage is very coarse-grained. Currently, we simply expect a safe subset of system calls to be used by non-core components. A more nuanced analysis of system calls and their arguments would leverage static analysis to aid in guaranteeing temporal separation. Second, in SafeFlow, we ignored value flow through *storage channels* such as a value written to disk or common buffer in the OS by a non-core component, which is read by the core component. Analyzing these flows will be require a fine-grained analysis of system call invocation statically and at run-time. Finally, while SafeFlow is effective at identifying unsafe values accessed within the core component, verifying that these unsafe values do not affect critical data in the core component can result in false positives. This is particularly true if the unsafe value propagates along a long path before it affects the critical data. While more accuracy in our value flow analyses can mitigate the number of false

positives, ranking errors in some priority order would be very helpful to the programmer in addressing errors in decreasing order of importance or accuracy.

Chapter 5

Conclusions

In this dissertation, we have motivated and established our broad goal of developing a programming environment supported by a suite of static and dynamic analysis tools, which verify that the implementation of robust embedded system architectures conform to the assumptions of architecture as well as the architectural design itself. We identified the main implementation errors, which violate the architectural assumptions and designs as being memory errors, logical system resource errors, timing errors, hidden dependencies due to inadvertent global variables, and missing architectural checks on some paths in the system. We have developed static analysis techniques to verify that these errors do not occur or prevent these errors from violating the architectural design and assumptions

In particular, we have developed Control-C, which is basically a type-safe subset of C through semantic restrictions, that guarantees memory safety statically without run-time software checks or garbage collection. Significantly, Control-C provides these guarantes, while allowing arbitrary dynamic memory allocation with explicit deallocation imposing memory consumption overheads only in a small fraction of the programs. Our experiments have shown that Control-C is expressive enough for a large number of control and embedded systems, and an even larger class of systems if we use run-time software checks for complex array accesses. Dhurjati's thesis [30] and our paper [31] has furthered this work and enabled guaranteeing memory safety with minimal run-time overhead and no garbage collection for practically the full generality of C.

We also defined the implementation errors that can cause erroneous dependencies of core components on non-core shared memory values in embedded systems. We have developed an annotation mechanism and accompanying static analyses to guarantee that critical components always monitor data from unreliable components before using them. Our analysis checks that every path contains the architectural monitoring checks and that there are no hidden dependencies through inadvertent shared variables accessible to the unreliable components. Our implementation has detected multiple hidden dependencies through an inadvertent shared variable in the different eSimplex implementation of the Simplex architecture.

Further, we developed a safe shared memory library, which enables development of future core components, which communicate with non-core components safely monitoring all incoming non-core values.

Finally, we have also developed a co-design based architecture to protect against timing delays in sensor updates and control outputs. Our architecture tolerates communication delays, and sensor and controller restarts. This architecture is very similar to safe value propagation and our techniques can be easily adapted to annotate timing assumptions and to analyze that all incoming feedback is filtered through a state estimator, which validates its timeliness.

All our tools can be integrated to provide a programming environment for robust embedded system implementation. In order to develop a comprehensive system, in addition to the above tools, we also require the assistance of light-weight dynamic monitoring, system utilities, formal methods or manual inspection, particularly to verify temporal isolation. We have demonstrated that static analysis reduces our dependence on these utilities, which are either unattractive for embedded systems (run-time overheads) or do not scale to real codes (many formal techniques or manual inspection). Through this work, we have shown that static analysis is a vital technique to verify architectural assumptions and high-level designs in the implementation.

Appendix A

Restrictive Control-C for control applications

We initially developed a restrictive version of Control-C, which was expressive enough for control applications [60]. In this section, we provide the language rules for this restrictive version. The Control-C language restrictions presented in 2 accept a superset of the programs accepted by *restrictive-Control-C*.

A.1 Basic Rules

The first set of rules focus on type-safety, uninitialized pointers, and some basic issues

(**T1**) The Control-C language requires strong typing of all functions, variables, assignments, and expressions, using the same types as in C.

(**T2**) The language disallows casts to or from any pointer type. Casts between other types (e.g., integers, floating point numbers, and characters) are allowed.

(**T3**) A union can only contain types that can be cast to each other; consequently a union cannot contain a pointer type.

(**T4**) The language requires that there are no uses of uninitialized local pointer variables within a procedure. In particular, a pointer variable must be assigned a value *before it is used or its address is taken*.

(**T5**) If the application chooses to exploit assumption (S3) above to avoid runtime NULL pointer checks, then any individual data object (scalar, structure, or array, allocated statically or dynamically) should be no larger than the size of the reserved address range.

(**T6**) Pointer arithmetic is disallowed in Control-C.

Explicit array declarations are just as in C, i.e., they need to specify the size of each dimension, except for the first dimension of an array formal parameter.

A.2 Array Indexing Rules

Array operations in Control-C must obey the following rules. On all control flow paths,

(**A1**) The index expression used in an array access must evaluate to a value within the bounds of the array.

(**A2**) For all dynamically allocated arrays, the size of the array has to be a positive expression.

(**A3**) If an array, `A`, is accessed inside a loop, then

(a) the bounds of the loop have be to be provably affine transformations of the size of `A` an outer loop index variables or vice versa;

(b) the index expression in the array reference, has to be provably an affine transformation of the vector of loop index variables, or an affine transformation of the size of `A`; and

(c) if the index expression in the array reference depends on a symbolic variable `s` which is independent of the loop index variable (i.e., appears in the constant term $\vec{q}$ in the affine representation), then the memory locations accessed by that reference have to be provably *independent* of the value of `s`.

(**A4**) If an array is accessed outside of a loop then

(a) the index expression of the array has to be provably an affine expression of the length of the array.

A.3 Regions and Dynamic Memory Allocation

The language provides three intrinsic functions (replacing `malloc` and `free`) for region-based memory management: `RInit`, `RFree`, and `RMalloc`. The region is made active by calling `RInit()` and freed (made inactive) using `RFree()`. `RMalloc` has the same signature as `malloc`, and allocates memory within the active region.

The constraints imposed by Control-C to allow safe dynamic memory allocation are summarized below.

(**R1**) Only one region is active at any point in the program. Thus calls to `RInit` and `RFree` must alternate on every potential path in the program, starting with an `RInit`.

(**R2**) The region should be active at any call to RMalloc.

(**R3**) Because region memory must not be accessed after a region is freed, any pointer value that contains a region address must be provably dead (unused or unreachable) at a call to `RFree`. To verify this statically we have the following constraints

(a) Any local and global scalar pointer variable must be explicitly reinitialized following a call to `RFree`, before any potential uses of the variable. Note in particular that this includes variables that *never* point to the heap (see the discussion below).

(b) Structures or arrays containing pointers must be allocated dynamically, either via `RMalloc` (on the heap) or via `alloca` (on the stack). In particular, aggregate objects containing pointers cannot live in global or local variables (with the exception of initialized constants).

Rules R3(a) and R3(b) can result in a severe run-time penalty since all globals and local variables need to be reinitialized at every call to `RFree`, irrespective of whether they point to region memory. In order to avoid this, most compilers can analyze conservatively through an alias analysis if a scalar pointer or a pointer in an aggregate typed variable hold an address within the heap (region). Thus, we evolve the following modifications to Rule R3, which may cause programs to be non-portable in the case that a compiler on a different platform does not have the ability to perform this alias analysis. However, we justify this non-portability by the potential peformance enhancement, which is important for real-time control systems. The weaker versions of R3(a) and R3(b) are the following:

(**R3**′(a)) Any local and global scalar pointer variable that may hold an address within a heap region must be explicitly reinitialized following a call to `RFree`, before any potential uses of the variable.

(**R3**′(b)) Pointer fields are permissible within global or local aggregate objects only if they provably never hold an address within a heap region, and their address is never taken. All other aggregate objects containing pointer fields must be allocated dynamically, either via `RMalloc`) or via `alloca`.

Appendix B

Architecture for Tolerance of Timing Delays

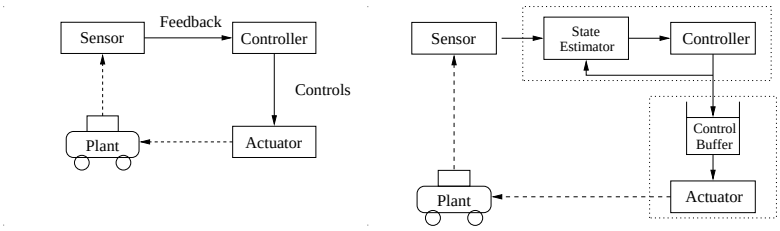
In chapter 1, we discussed architectures such as Simplex, which enable system robustness in the presence of value errors, timing errors, and runtime errors. In this chapter, we describe a robustness architecture, which tolerates delays in networked control systems through a co-design architecture. The architecture significantly extends deadlines for sensor feedback and controller updates enabling controllers and actuators to operate reliably even in the presence of delays due to communication or component restarts. This enables controllers and actuators achieve *local temporal autonomy* [46], which states that a component should be able to function autonomously and reliably for a certain period of time in the presence of peer component failures.

B.1 Problem

Networked control systems are conventionally designed with periodic components with hard real-time guarantees. Sensors generate periodic updates, controllers use these updates to compute periodic controls, actuators implement these controls periodically, and most or all of these operations are completed within hard deadlines. This mode of operation significantly simplifies control design, and has consequently been enforced in most control systems. To enable this approach, systems engineers have relied on effective scheduling algorithms based on conservative worst-case execution deadlines, and communication channels such as CAN [37] and FDDI [2] that provide real-time guarantees.

In such tightly coupled systems, missed deadlines in sensor updates or controller computations usually result in the system transitioning to a *fail-safe* state. This reduces system performance significantly, and hence the system is usually over-engineered to ensure that deadlines are met. Consequently, the correctness of system operation depends strongly on the periodicity of sensor updates and controller computations.

With the widespread proliferation of best effort networks such as Ethernet and 802.11 [76], control systems using such networks are being increasingly deployed. However, such networks cannot provide real-time guaran-



(a) Original design (b) Enhanced design

Figure B.1: Design enhancements in a typical control loop

tees, and wireless networks in particular have unpredictable delays and high packet losses. This clearly violates the assumptions of digital control design, and designing such systems using conventional techniques will result in the system frequently entering the fail-safe state.

Further, transient errors and residual software bugs in individual components, particularly commercial off-the-shelf (COTS) components, often result in component failures and software crashes. Such failures are usually handled by state check-pointing and component restarts. In fact, software rejuvenation [95] is a well-documented fault management technique where components are pro-actively restarted to avoid severe crash failures in the future. In addition, planned software upgrade of components also involves replacing an existing component. However, all these changes may result in deadline violations necessitating fail-safe action. Hence, extension of deadlines and graceful system degradation are imperative to improving system performance during such changes.

B.2 Co-design Based Architecture

Networked control systems are characterized by components operating in control loops. A typical control loop, shown in figure B.1(a), consists of sensors, controllers, and actuators that control a physical plant. Sensors monitor physical characteristics in the plant, and send periodic updates as feedback to the controller. The controller uses this feedback, and computes controls to achieve prescribed goals. These are then sent to actuators that enforce the controls in the plant.

Digital control design assumes that all components in a control loop operate periodically with hard real-time deadlines. For every period, the controller receives feedback from the sensor and computes a set of controls that are enforced by the actuator. The computation deadlines and communication delays have strict guarantees. In particular, violation of a controller deadline would result in no control being sent to the actuator. This usually results in a fail-safe operation being enforced by the actuator. Similarly, a missed feedback deadline by the sensor would result in the controller miss-

ing its deadline, and leads to fail-safe action as well.

In networked control systems, such deadline violations can occur due to: (a) communication delays and data losses between the components, or (b) failures and consequent restarts of the sensor or the controller. Communication delays on the link from the sensor to the controller can be partly addressed by using a state estimator at the controller. However, deadline misses due to delays on the link from the controller to the actuator, or component failures, are harder to address. We do not consider actuator failures since they are usually simple and robust components, and are separated from the complex functionality of the controller for restart independence.

As noted above, controllers usually have a state estimator that maintains a *model* of the physical plant. This model captures the behavior of the plant by representing its current configuration with a set of variables called the *state*. The state in the model evolves according to the controls applied to the plant through the actuator. However, imprecise modeling and imperfect calibration lead to an error between the states of the model and the plant. This error is corrected by feedback from the sensors.

Our approach is based on two observations about this mechanism. First, control systems usually have certain tolerances that allow the plant to be operated within specified error bounds. For instance, a car in our testbed can be off by a couple of inches without causing any collisions. Second, the state estimator has a predictive capability, in that, its state can evolve based only on the controls. Hence, as long as the error between the estimated and actual states can be bounded, the plant can be guaranteed to operate within the error bounds specified by system tolerance.

The above observations can be used to enhance the control loop as shown in figure B.1(b). Specifically, we introduce the following enhancements:

I. The controller uses a *state estimator* to tolerate delays in sensor updates.

II. The controller computes a *sequence* of future controls that are stored in a *control buffer* at the actuator.

The first enhancement ensures that controllers have periodic estimates of plant state from the state estimator. This allows controllers to be designed using traditional digital control theory, and bridges the gap between real-time control and best effort networks. For the second enhancement, the controller computes a sequence of future controls, instead of just a single control for the current period. To accomplish this, the estimator is used as a state predictor to estimate the state of the plant as the sequence of future controls is applied to it. These future controls are then stored in the control buffer at the actuator, and are used in case the controller misses future deadlines. Note that the second enhancement is what allows deadlines to be extended in the sensor and the controller.

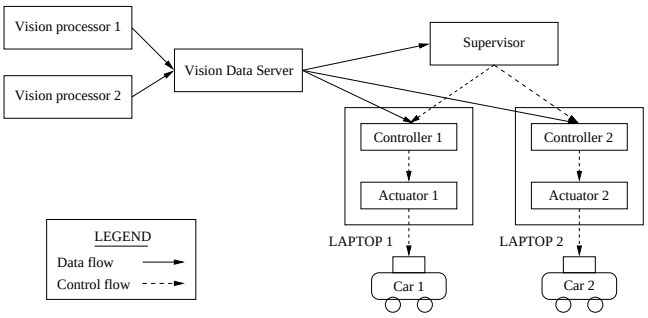


Figure B.2: Software architecture of the testbed

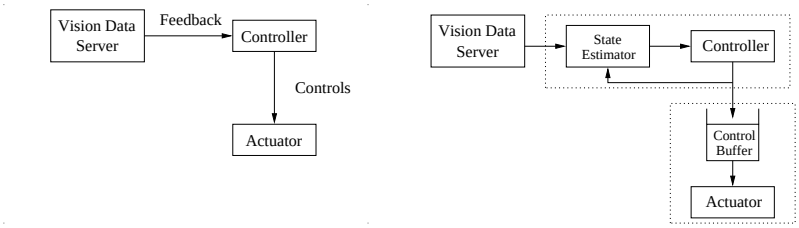
The key idea in our approach is to reduce dependencies between components that communicate over a network. However, the performance gain is primarily determined by the horizon for which future controls are computed, since the horizon determines extensions in deadlines. In particular, the future control horizon depends on the following factors:

1. System tolerance and operational error bounds for the plant.
2. Growth of error in predictions by the state estimator.
3. Computational resources available to the controller to compute a sequence of future controls during each period.
4. Size of the control buffer at the actuator to store the future controls.
5. Communication bandwidth between the controller and the actuator to send a sequence of future controls during each period.

The last three factors depend on system deployment constraints and can be engineered as necessary. However, the growth of prediction error is determined by the plant model, and the future control horizon is essentially the interval up to which this error is within system tolerance. Thus, in section B.4, we will only consider the the system tolerance and the prediction error in order to estimate the deadline extension achieved by this technique

B.3 Application to Cars Testbed

The software architecture of the traffic control testbed shown in figure 1.2 is illustrated in figure B.2. There are two video processor components that process video streams from corresponding cameras to determine car positions and orientations. This information is accumulated and merged in a vision data server component, and is then available as feedback to other components in the system. The controller and actuator components for each car execute on the corresponding laptop as shown in the figure. All



(a) Original design (b) Enhanced design

Figure B.3: Design enhancements in lower level control loop

other components execute on separate computers and communicate over a network.

There are two control loops in the system as outlined in figure B.2. The lower level loop involves the controller and actuator components that control corresponding cars. Each controller takes in a trajectory, which is a timed sequence of track positions, and computes a sequence of controls to operate the car along the given trajectory. The controls are sent to the actuator, which then transmits them to the car on the dedicated RF channel.

The higher level control loop involves the supervisor component, which computes desired car trajectories and sends them to the controller. For instance, in a traffic scenario the computed trajectories move the cars to their destinations along a network of roads. If correctly followed, the trajectories also ensure that cars do not collide, particularly at road intersections [43]. Hence, the main problem in the lower level control loop is to follow these trajectories as closely as possible.

B.3.1 Design Enhancements

We have applied the ideas of section B.2 to enhance the software architecture of the testbed and improve system robustness. In this work, however, we only consider design enhancements for the lower level control loop in figure B.2.

The key components and connections of the lower level control loop are reproduced in figure B.3(a). As shown in the figure, the controller *depends* on the vision data server, since the controller would have to operate with obsolete data if any updates are lost over the network. Similarly, the actuator *depends* on the controller as it can only transmit the last received control to the car. Consequently, this design is not robust, as updates from the vision data server and the controller have strict deadlines, and component failures are hard to address.

These problems are similar to those illustrated in figure B.2, and the same solutions apply as shown in figure B.3(b). The controller uses a state estimator to get periodic estimates and tolerate delays in sensor updates. The state estimator is also used as a predictor to compute a sequence of

future controls that are stored in the control buffer at the actuator. In particular, the extension of deadlines for updates from the sensor and the controller is determined by the accuracy of state prediction. This issue is analyzed in detail in the next section.

B.4 Analytical and Experimental Validation

We validate the above technique through theoretical analysis and verify these analyses empircally [59]. While we present the validation of our approach for completeness, it is discussed in detail in Girish Baliga's dissertation. We only summarize the results here. We address the following questions:

1. What is the theoretical deadline extension achievable by this design?
2. Does the empirical growth in estimation error in the absence of feedback correspond to our theoretical extensions?
3. Is the testbed able to tolerate sensor and controller restarts (with varying restart times) safely?

B.4.1 Analysis of State Prediction

The detailed analysis of the error growth in state prediction due to the state estimator has been discussed in [59]. We modeled the error in the state of the car as error in its position (x,y) and orientation(θ). The equation below represents the transition equation which calculates the state of a car at time $t+1$, given its state at time t.

$$\mathbf{x}_{t+1} = \mathbf{M}_t \mathbf{x}_t + \mathbf{w}_t \tag{B.1}$$

where $\mathbf{x}_t$ is the *state* of the car, $\mathbf{M}_t$ is the *car model*, and $\mathbf{w}_t$ is the *update error*, all at time t.

In the absence of feedback, the prediction error $\tilde{\mathbf{x}}_{t+1}$ grows with time, giving us the following equation:

$$\tilde{\mathbf{x}}_{t+1} \leq \mathbf{w}_{max} t \tag{B.2}$$

where $\mathbf{w}_{max} t$ is the maximum prediction error in any step. Thus, in the worst case the prediction error grows linearly with time with a constant factor bounded by the maximum error in each time step..

However, in the mean error bound analysis, we have shown that the mean prediction error grows as the square root of time given by:

$$\epsilon = \sqrt{E[\tilde{\mathbf{x}}_{t+1}^T \tilde{\mathbf{x}}_{t+1}]} = k\sqrt{t} \tag{B.3}$$

for constant k. In practice, this equation gives a much longer deadline extension compared to equation (B.2).

B.4.2 Experimental Results

As mentioned above, the experimental results are discussed in detail in our paper [59] and in Girish Baliga's dissertation. They are outlined here for completeness. We implemented our architecture on Etherware [10, 11], a message-oriented component middleware for networked control systems. Etherware components interact using messages and have their own message queues in which incoming messages are stored. Etherware provides efficient restart mechanisms for components, by checkpointing component state and re-initialization. For our experiments, sensors and controllers on Etherware were restarted after varying intervals of time to study the effect of delays on the behavior of the cars. The model used by each car controller in the testbed implementation is a reasonable calibration of the discrete speeds and steering angles of the car. This model functions as a state predictor. The actuator is implemented to contain a control buffer to hold the sequence of speculative controls overwritten during each period, and used in the event of delays.

Our experiment was that of two cars in a motorcade, with the follower car closely following the leader car separated by 175mm 'bumper-to-bumper' along an elliptical trajectory with major axis 2.8m and minor axis 2m. The cars themselves are 225mm long and travel at an average speed of 371mm per second. Each loop around the oval takes 21.7s to complete. Cars are controlled autonomously. The safety constraint is that the maximum deviation of a leader car from its assigned trajectory should be 50mm. The sensor and controller operate with 100ms periods.

We measured the growth of the state predictor error in the absence of feedback by comparing the state to the actual state of the car. The error is simply measured to be the distance (ignoring the orientation). Figure B.4 plots the error as a function of time elapsed since the last feedback was received by the controller (dotted and dashed lines). The many such lines represent the experiment repeated at different points in the trajectory. The figure also shows the representative curve, approximately $y = ksqrt(x)$, where k is 1.0769, the function with the least mean squared error from the prediction curves. This agrees with our average case analysis in equation (B.3). Also, the maximum difference between calibrated speeds is 127mm/s. Assuming this to be the worst case error in eq (B.2) the dashed line shows the max error growth over a 2 second period from the time last feedback was received. The average error and the maximum error provide us the extended average-case bound and the worst-case bound for the new real-time deadline for the sensors and controllers due to our co-design architecture.

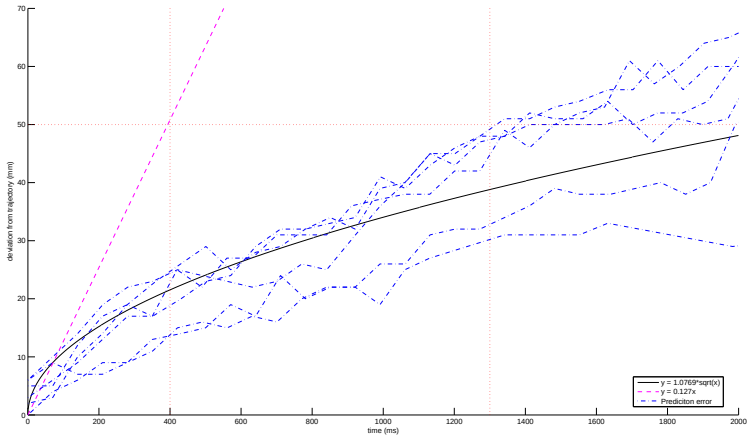


Figure B.4: Growth of error in state prediction

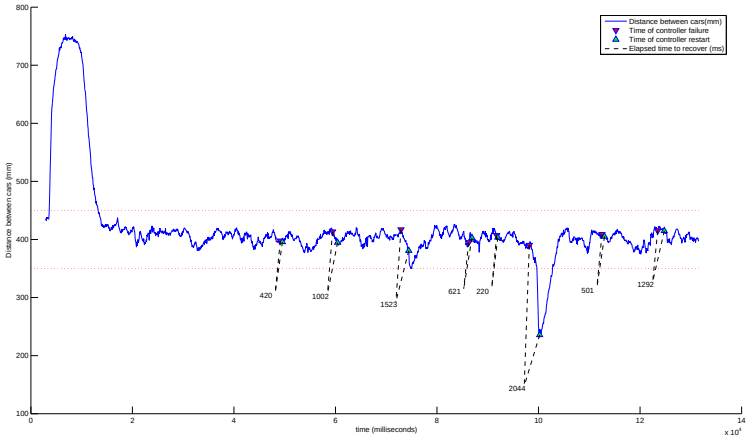


Figure B.5: Distance between centers of cars in the motorcade

These are calculated by bounding the maximum deviation at 50mm, thus giving us a 1300ms bound in the average case and a 400ms bound in the worst case. This is a significant increase over the 100ms period of the sensor and the controller, which permits communication delays and arbitrary component restarts.

The second experiment consisted of sporadically restarting the controller of the leader car with varying restart times, thus simulating restarts and delays or a series of packet losses in the network. In rder to evaluate the effectiveness of our mechanism, we disable the fail-safe in the system and identify the delays for which the deviation of each car grows beyond 50ms. We are primarily interested in the deviation of the leader car from its trajectory. The minimum distance between the two cars (equal to the distance

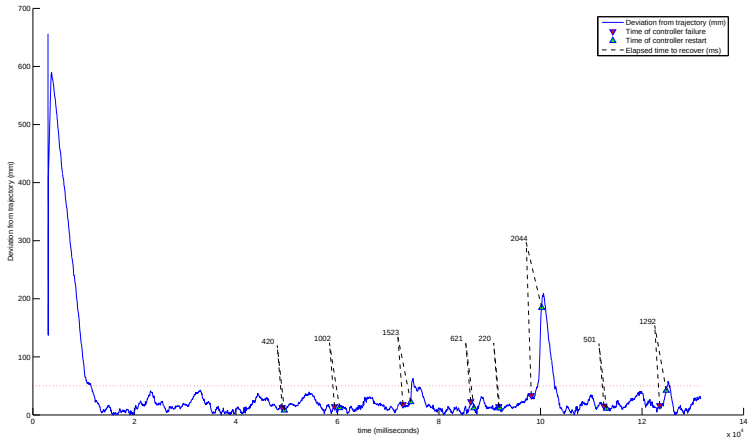


Figure B.6: Deviation of leader from its trajectory

between the centers of the two cars) is equal to the length of a car, 225mm. The plot in figure B.5 demonstrates the physical manifestation of restarts at different points on the trajectory and its temporal characteristics while using our codesign architecture. The corresponding changes in the distance between the cars is shown in figure B.6. The restart time in milliseconds each case is shown in the graph. Inherent noise in the measurement, actuators, and trajectory computation causes the car to deviate by 25-35mm even under normal operation.

In our architecture, we determined that 1300ms was the average case upper bound for the future control horizon. We use a horizon of 1200ms for our experiments. A 'STOP' command is appended at the end of the horizon of future controls. The initial deviation in the leader car until it catches up is due the difference between the actual position of the leader car and its assumed position in the trajectory. We begin restarting the leader car controller after allowing the cars to stabilize over one elliptical loop (21.7s). We observe that the deviation of the leader car exceeds the 50mm safety bound only when restart time is greater than 1200ms. The restart time of 2044ms, results in the largest deviation for the leader car and the minimum distance between the cars due to a collision. Restarts within 1200ms are unnoticeable with perturbations within the maximum deviaition of 50mm. Upon recovery, the leader car gains speed and catches up with its desired trajectory and stabilizes the deviation.

B.5 Related Work

The problem of tolerating delays and errors in sensor feedback has been addressed from a control theoretic perspective in previous work. For instance,

the effect of such delays on the operation and stability of a controller has been studied in [70] and [96], and Nilsson [75] has analyzed the use of a state estimator to stabilize controllers in the presence of random delays. However, the focus of such research has been to improve the performance of control algorithms in the presence of delays in sensor updates. In particular, software robustness issues based on real-time deadline extensions, and tolerance to changes such as component restarts and upgrades have not been addressed. There are many other robustness architectures, which protect against value errors, software bugs including Simplex and other works from section 1.6.1. Restarts of individual components for recovery, pro-actively as well as reactively, have been described in software rejuvenation [95] and recovery-oriented computing [41, 14] techniques.

B.6 Chapter Summary

The key contributions in this chapter are summarized below:

1. We present a co-design based approach to improve robustness in networked control systems. Our modular approach allows the control engineer to use the traditional periodicity assumptions, and the systems engineer to build a system tolerant to communication delays and component restarts.

2. We describe how our approach has been used to significantly improve robustness in a traffic control testbed, a prototype networked control system.

B.7 Future Work

We have established a co-design based architecture to enable local temporal autonomy for actuators and controllers in a networked control system. In general, timing assumptions made by a component should be verified to be satisfied along all the paths in the component. In particular, different components in a controller, such as the control computation, the safety envelope calculation, etc. require feedback with precision and delay bounds. The timing requirements of each of these functions should be satisfied along all paths through the system. In the presence of the state estimator, the ability of the estimator to tolerate delays in the peer component should be specified.

Bibliography

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] ALBERT, B., AND JAYASUMANA, A. P. *FDDI and FDDI-II: Architecture, Protocols, and Performance*. Artech House Publishers, Jan 1994.

[3] ALEPH ONE. Smashing the stack for fun and profit. *Phrack 7*, 49 (November 1996).

[4] AMER DIWAN, HAN LEE, D. G., AND FARKAS, K. Energy consumption and garbage collection in low-powered computing. Tech. Rep. CU-CS-930-02, University of Colarado-Boulder, 1996.

[5] ARORA, A., AND KULKARNI, S. S. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Trans. Softw. Eng. 24*, 6 (1998), 435–450.

[6] AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 290–301.

[7] AVIZIENIS, A. *Software Fault Tolerance*. John Wiley and Sons, NY, 1995, ch. The Methodology of N-Version Programming.

[8] B. A. CARR, J. R. G. Spark - an annotated ada subset for safety-critical programming. In *Proceedings of Tri-Ada Conference* (Dec 1990).

[9] BACON, D., CHENG, P., AND RAJAN, V. A real-time garbage collector with low overhead and consistent utilization. In *Proc. 30th ACM Symp. Principles of Programming Languages (POPL03)* (Jan. 2003), pp. 285–298.

[10] BALIGA, G., GRAHAM, S., SHA, L., AND KUMAR, P. R. Etherware: Domainware for wireless control networks. In *Proc. of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2004)* (Vienna, Austria, May 2004), pp. 155–162.

[11] BALIGA, G., GRAHAM, S., SHA, L., AND KUMAR, P. R. Service continuity in networked control using etherware. *IEEE Distributed Systems Online* (Sep 2004).

[12] BOLLELLA, G., AND GOSLING, J. The Real-Time specification for Java. *Computer 33*, 6 (2000), 47–54.

[13] BOYAPATI, C., SALCIANU, A., BEEBEE, W., AND RINARD, M. Ownership types for safe region-based memory management in Real-Time Java. In *SIGPLAN Conference on Programming Language Design and Implementation* (2003), pp. 324–337.

[14] CANDEA, G., CUTLER, J., FOX, A., DOSHI, R., GARG, P., AND GOWDA, R. Reducing recovery time in a small recursively restartable system. In *Proceedings of the International Conference on Dependable Systems and Networks* (Washington, D.C., June 2002).

[15] CANDEA, G., DELGADO, M., CHEN, M., AND FOX, A. Automatic failure-path inference: A generic introspection technique for internet applications. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications* (Washington, DC, USA, 2003), IEEE Computer Society, p. 132.

[16] CANDEA, G., AND FOX, A. Recursive restartability: Turning the restart sledgehammer into a scalpel. In *Proc. HOTOS-VIII* (Schloss Elmau, Germany, 2001).

[17] CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. Modula3 language definition. *ACM Sigplan Notices 27*, 8 (Aug. 1992).

[18] CHEN, F., AND ROSU, G. Java-mop: A monitoring oriented programming environment for java. In *TACAS* (Oct 2005).

[19] CHERNYSHOV, M. Post-mortem on failure. *Nature 339*, 9 (May 1989).

[20] CHIN, W.-N., CRACIUN, F., QIN, S., AND RINARD, M. Region inference for an object-oriented language. *SIGPLAN Not. 39*, 6 (2004), 243–254.

[21] CHUI, C. K., AND CHEN, G. *Kalman filtering with real-time applications*. Springer-Verlag, 1999.

[22] CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. Ccured in the real world. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation* (June 2003), pp. 232–244.

[23] COOK, A., AND HUNT, K. Arinc 653-achieving software re-use. *Microprocessors and Microsystems 20*, 8 (Apr 1997), 479–83.

[24] COPPOT, D., AND SULLIVAN, K. J. Galileo: a tool built from mass-market applications. In *International Conference on Software Engineering* (2000).

[25] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas* (New York, NY, 1999), pp. 262–275.

[26] CUNHA, J., AND RELA, M. On the use of disaster prediction for failure-tolerance in feedback control systems. In *International Conference on Dependable Systems and Networks* (Washington D.C., USA, June 2002).

[27] DAS, M., LERNER, S., AND SEIGLE, M. Esp: Pathsensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), pp. 57–68.

[28] DELINE, R., AND FAHNDRICH, M. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation* (Snowbird, UT, June 2001), pp. 59–69.

[29] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (1977), 504–513.

[30] DHURJATI, D. Safecode: Enabling sound static analyses for weakly typed languages. Thesis proposal, University of Illinois at Urbana-Champaign, 2005.

[31] DHURJATI, D., KOWSHIK, S., AND ADVE, V. Enforcing alias analysis for weakly typed programs. In *Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation* (2006), p. To Appear.

[32] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without runtime checks or garbage collection. In *LCTES* (San Diego, CA, Jun 2003), pp. 69–80.

[33] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without garbage collection for embedded applications. *Trans. on Embedded Computing Sys. 4*, 1 (2005), 73–111.

[34] DING, H., AND SHA, L. Dependency algebra: A tool for designing robust real-time systems. In *RTSS* (2005), pp. 210–220.

[35] ENGLER, D. R., AND ASHCRAFT, K. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP* (2003), pp. 237–252.

[36] ET AL, J.-L. L. Ariane 5 flight 501 failure. Tech. rep., ESA Press Realease, Paris, France, 1996.

[37] ETSCHBERGER, K. *Controller Area Network*. IXXAT Automation GmbH, Aug 2001.

[38] FAHNDRICH, M., AND DELINE, R. Adoption and focus: Practical linear types for imperative programming. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (June 2002), pp. 13–24.

[39] FEILER, P., LEWIS, B., AND VESTAL, S. The sae avionics architecture description language (aadl) standard. In *RTAS Workshop on Model-driven Embedded Systems* (2003).

[40] GAY, D., AND AIKEN, A. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 313–323.

[41] GEORGE CANDEA, A. F. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (Lihue, HI, May 2003).

[42] GEORGE CANDEA, JAMES CUTLER, A. F. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal 56*, 1-3 (March 2004).

[43] GIRIDHAR, A. Scheduling traffic on a road network. Master's thesis, University of Illinois at Urbana-Champaign, December 2002.

[44] GORDON, A. D., AND SYME, D. Typing a multi-language intermediate code. *ACM SIGPLAN Notices 36*, 3 (2001), 248–260.

[45] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification*. Sun Microsystems, 2000.

[46] GRAHAM, S. *Issues in the convergence of control with communication and computation*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 2004.

[47] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation* (June 2002), pp. 282–293.

[48] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization* (Austin, TX, Dec. 2001), pp. 1–12.

[49] HAVELUND, K., AND ROSU, G. Efficient monitoring of safety properties. *STTT 6*, 2 (2004), 158–173.

[50] HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Software verification with blast. In *Tenth International Workshop on Model Checking of Software (SPIN)* (2003), pp. 235–239.

[51] HENZINGER, T. A., AND KIRSCH, C. M. The embedded machine: Predictable, portable real-time code. In *Proc. SIGPLAN '02 Conf. on Programming Language Design and Implementation* (Berlin, Germany, June 2002).

[52] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for network sensors. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2000), pp. 93–104.

[53] J.C. CUNHA, R. MAIA, M. R., AND SILVA, J. A study on the failure models in feedback control systems. In *International Conference on Dependable Systems and Networks* (Goteborg, Sweden, July 2001).

[54] JHUMKA, A., HILLER, M., AND SURI, N. An approach for designing and assessing detectors for dependable component-based systems. In *HASE* (2004), pp. 69–78.

[55] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of c. In *Proc. USENIX Annual Technical Conference* (June 2002), pp. 275–288.

[56] JONES, R. W. M., AND KELLY, P. H. J. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging* (1997), pp. 13–26.

[57] KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHPEISMAN, T., AND WONNACOTT, D. The Omega Library Interface Guide. Tech. rep., Computer Science Dept., U. Maryland, College Park, Apr. 1996.

[58] KIM, M., LEE, I., SAMMAPUN, U., SHIN, J., AND SOKOLSKY, O. Monitoring, checking, and steering of real-time systems. *Electr. Notes Theor. Comput. Sci. 70*, 4 (2002).

[59] KOWSHIK, S., BALIGA, G., GRAHAM, S., AND SHA, L. Co-design based approach to improve robustness in networked control systems. In *DSN* (2005), pp. 454–463.

[60] KOWSHIK, S., DHURJATI, D., AND ADVE, V. Ensuring code safety without runtime checks for real-time control systems. In *Proc. 2002 Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Grenoble, Oct 2002).

[61] LAMPSON, B. W. A note on the confinement problem. *Commun. ACM 16*, 10 (1973), 613–615.

[62] LATTNER, C., AND ADVE, V. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Nov 2003.

[63] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Progra m Analysis and Transformation. In *CGO* (San Jose, USA, Mar 2004).

[64] LATTNER, C., AND ADVE, V. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language* (Chicago, IL, June 2005).

[65] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture* (1997), pp. 330–335.

[66] LEE, C.-G., AND SHA, L. Real-time virtual machines for avionics software migration. *International Journal of Embedded Systems* (2006), To appear.

[67] LEE, K. Criticality-driven real-time recovery in a virtual machine environment. Thesis proposal, University of Illinois at Urbana-Champaign, 2005.

[68] LEVESON, N. G., CHA, S. S., KNIGHT, J. C., AND SHIMEALL, T. J. The use of self checks and voting in software error detection: An empirical study. *IEEE Trans. Software Eng. 16*, 4 (1990), 432–443.

[69] LEVESON, N. G., CHA, S. S., AND SHIMEALL, T. J. Safety verification of ada programs using software fault trees. *IEEE Softw. 8*, 4 (1991), 48–59.

[70] LING, Q., AND LEMMON, M. Soft real-time scheduling of networked control systems with dropouts governed by a markov chain. In *American Control Conference* (Denver, CO, June 2003).

[71] MYERS, A. C., NYSTROM, N., ZHENG, L., AND ZDANCEWIC, S. Jif: Java information flow, 2001.

[72] NECULA, G. C. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)* (Paris, Jan. 1997), pp. 106–119.

[73] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. In *Proc. 29th ACM Symp. Principles of Programming Languages (POPL '02)* (London, Jan. 2002), pp. 128–139.

[74] NILSSON, A., EKMAN, T., AND NILSSON, K. Real java for real time - gain and pain. In *CASES* (2002), pp. 304–311.

[75] NILSSON, J. *Real-time control systems with delays*. PhD thesis, Lund Institute of Technology, 1998.

[76] O'HARA, B., AND PETRICK, A. *The IEEE 802.11 Handbook: A Designer's Companion*. IEEE, Dec 1999.

[77] PETER FEILER, B. L., AND VESTAL, S. The sae aadl standard: A basis for model-based architecture-driven embedded systems. In *Workshop on Model-Driven Embedded Systems, Real-Time Application Systems (RTAS) Conference* (2003).

[78] RINARD, M., CADAR, C., DUMITARN, D., ROY, D. M., LEU, T., AND JR., W. S. B. Enhancing server availability and security through failure-oblivious computing. In *OSDI* (Dec 2004).

[79] RUSHBY, J. Partititioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. rep., SRI International, Mar 1999.

[80] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.

[81] SHA, L. Dependable system upgrades. In *Proceedings of IEEE Real Time System Symposium* (1998).

[82] SHA, L. Using simplicity to control complexity. *IEEE Software* (July/August 2001).

[83] SHA, L., RAJKUMAN, R., AND GAGLIARDI, M. Evolving dependable real-time systems. Tech. Rep. CMS/SEI-95-TR-005, CMU, 1995.

[84] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium* (2001), pp. 201–220.

[85] SIEWIOREK, D. P., AND SWARZ, R. S. *Reliable Computer Systems: Design and Evaluation*, 3rd edition ed. AK Peters, Ltd., 1998.

[86] STOBIE, K. Too darned big to test. *Queue 3*, 1 (2005), 30–37.

[87] TIRUMALA, A., CRENSHAW, T., SHA, L., BALIGA, G., KOWSHIK, S., ROBINSON, C., AND WITTHAWASKUL, W. Prevention of failures due to assumptions made by software components in real-time systems. *ACM SIGBED Review 2*, 3 (July 2005).

[88] TOFTE, M., AND BIRKEDAL, L. A region inference algorithm. *ACM Trans. Prog. Lang. Sys. 20*, 1 (1998), 724–767.

[89] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation* (Feb. 1997), 132(2):109–176.

[90] VOLPANO, D. M., AND SMITH, G. A type-based approach to program security. In *TAPSOFT* (1997), pp. 607–621.

[91] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (December 1993), 203–216.

[92] WALKER, D., AND MORRISETT, G. Alias types for recursive data structures. *Lecture Notes in Comp. Sci. vol. 2071* (2001), 177+.

[93] WALL, L., AND SCHWARTZ, R. L. *Programming Perl*. O'Reilly, 1991.

[94] WEIMER, W., AND NECULA, G. C. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)* (Oct. 2004), pp. 419–431.

[95] Y. HUANG, C. KINTALA, N. K., AND FULTON, N. Software rejuvenation: analysis, module and applications. In *Proc. of the 25th Int. Symposium on Fault-Tolerant Computing* (Pasadena, CA, June 1995).

[96] ZHANG, W. *Stability analysis of networked control systems*. PhD thesis, Case Western Reserve University, 2001.

Author's Biography

Sumant Kowshik is a Ph.D in computer science from the University of Illinois at Urbana-Champaign with a broad interest in building usable, reliable, and secure software. His main focus has been on building tools based on static analysis and software engineering principles to detect architectural design violations in embedded systems. In addition, he has also worked on easing the efforts of generic software developers through bug-detection and other analysis tools. His publications have appeared in leading conferences and journals in embedded systems, compilers, and dependability. Apart from technology, he has also had contributed to various developmental activities, business consulting engagements, and in organizational roles. He received his undergraduate degree in computer science from IIT Madras in India.